UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Grado en Ingeniería del Software

Control automatizado de un acuario de peces mediante Arduino

Automated control of a fish tank via Arduino

Realizado por
Miguel Ángel González Valadez

Tutorizado por
Luis Manuel Llopis Torres
Vicente Jesús Benjumea García

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, diciembre de 2023

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
INFORMÁTICA
GRADO EN INGENIERÍA DEL SOFTWARE

Control automatizado de un acuario de peces mediante Arduino

Automated control of a fish tank via Arduino

Realizado por
Miguel Ángel González Valadez

Tutorizado por
Luis Manuel Llopis Torres
Vicente Jesús Benjumea García

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, DICIEMBRE DE 2023

Resumen del Trabajo de Fin de Grado

Español

Este Trabajo de Fin de Grado se ha realizado por Miguel Ángel González Valadez bajo la tutorización de los profesores Luis Manuel Llopis Torres y Vicente Jesús Benjumea García. Este **TFG** se titula *Control automatizado de una pecera mediante Arduino* y consiste en cómo hacer un control casero de una pecera con un **Arduino UNO**, que controlará ciertos parámetros de la pecera (**pH**, **temperatura del agua**...) mediante un **servidor MQTT** y una **aplicación móvil** diseñada con *Xamarin Forms*.
También, se ha implementado un pequeño **cliente MQTT** hecho en `Python` con el fin de ver la transmisión de datos del Arduino al servidor MQTT y viceversa con sus respectivos *topics* y mensajes.

Palabras clave: **Arduino**, **MQTT**, **Xamarin Forms** y **Pecera**

English

This End-Of-Degree Thesis was done by Miguel Ángel González Valadez under the tutelage of professors Luis Manuel Llopis Torres and Vicente Jesús Benjumea García. This **EDT** is called *Automated control of a fish tank via Arduino* and consists in how to make a homemade control of a fish tank with an **Arduino UNO**, that will control some measurements (like **pH**, **water temperature** ...) through a **MQTT server** and a **mobile application** powered by *Xamarin Forms*.
Also, a small `Python` **MQTT client** was implemented in order to watch the data transfer from Arduino to the MQTT server and viceversa with their respective topics and payloads.

Keywords: **Arduino**, **MQTT**, **Xamarin Forms** and **Fish tank**

Índice

1. Introducción

En este proyecto titulado *Control automatizado de una pecera mediante Arduino*, se explicará de forma concisa y clara cómo hacer un controlador de peceras utilizando un Arduino UNO, algo de electrónica básica, un intermediario que servirá para transmitir datos de forma continua e inalámbrica (en este caso, un **bróker MQTT**), y una aplicación móvil de Android para informar al usuario de los **parámetros actuales** en todo momento, poder actualizar los **ajustes de referencia de la pecera** (como la temperatura deseada, establecer un rango de pH óptimo, entre otros), y poder recibir notificaciones push en caso de que algún **ajuste deseado** exceda de algún rango de tolerancia determinado.

1.1. Motivación y objetivos

Este trabajo está motivado por la necesidad de controlar una pecera de forma recurrente y sin necesidad de estar físicamente al lado de ella; el usuario podría estar haciendo otros quehaceres y no tiene el tiempo de estar vigilándola.
Se desea vigilar la **temperatura**, la **acidez** del agua (**pH**), la **profundidad** o **nivel** del agua y **la hora actual** del sistema. También, se desea controlar ciertos ajustes de la pecera, como la **temperatura deseada**, el **horario de encendido de la luz**, cuándo se encenderá el **comedero** diariamente, el **rango de pH** del agua y el **nivel mínimo** y **máximo** permitido de agua, mediante *notificaciones push*, que informarán al usuario de cuándo un ajuste de los anteriormente mencionados se sale del rango deseado.
Finalmente, se podría tener la necesidad de guardar el pH, la temperatura, la profundidad a una hora determinada en una **base de datos** y poder consultarlos y editarlos cuando se desee.

1.2. Estructura

La memoria se estructurará de la siguiente manera:

- Antecedentes: Descripción de los trabajos previos y conceptos en los que está basado este TFG.
- Tecnologías: Descripción de las tecnologías de las herramientas utilizadas.
- Metodología: Explicación de cómo se han llevado a cabo las reuniones, descripción de los *sprints* y cronología del trabajo realizado.
- Análisis del problema: Explicación del problema a resolver, con sus requisitos funcionales y no funcionales, y diagramas de casos de uso.
- Solución: Descripción minuciosa de la solución propuesta al problema detallado.
- Implementación y pruebas: Documentación de la implementación y pruebas de las partes del código más relevantes.
- Conclusiones: Algunas conclusiones que se podrían extraer del trabajo para ideas futuras relacionadas con el control automatizado de peceras.

- Bibliografía: Fuentes relevantes de información para profundizar acerca del contenido de este trabajo.
- Anexo: Manuales de instalación e información adicional que podría ser relevante.

2. Antecedentes

La base de todo este trabajo se fundamenta en una plataforma de desarrollo de *hardware* de código abierto para desarrollar prototipos electrónicos de forma rápida y sencilla. En este caso, se está utilizando Arduino, porque posee un microcontrolador programable que es muy versátil para proyectos de electrónica de complejidad media y alta.
Existen varios modelos de esta plataforma de desarrollo [1], como el Arduino UNO R3, Arduino UNO R4, que tiene conectividad Wifi para proyectos que requieran conexión al *Internet of Things*, entre otros.

Mantener manualmente un acuario requiere tiempo y dinero, porque hay que encargarse de vigilar que el nivel, temperatura y acidez del agua sean los correctos, mantener alimentados a los peces, oxigenar el agua para evitar que mueran asfixiados.
Es importante destacar que existen varias formas de **automatizar** un acuario, como por ejemplo, utilizando un comedero con un reloj programable, ya sea vía Wifi o por un panel de control para alimentar periódicamente a los peces, controlar varios dispositivos mediante una aplicación móvil ya creada. También, hay otros proyectos realizados con Arduino, que podrían ser de referencia para otros proyectos similares a este. [2, 3]

Para la parte de la aplicación, se podría utilizar algún protocolo de mensajería ligero y sencillo, ya que se quiere transmitir órdenes sencillas, como cambiar algún ajuste de referencia de la pecera, encender o apagar algún dispositivo de esta..., como el **protocolo MQTT**, ideal para redes no seguras y limitadas en cuanto a ancho de banda.

3. Tecnologías y herramientas utilizadas

En esta sección del trabajo, se describirá con detalle cada herramienta o tecnología utilizada para realizar el proyecto.

3.1. Arduino UNO R3

Es una placa de desarrollo de hardware basado en el microcontrolador `ATMega328p` [1], que es un miembro de lo que se denomina una familia de microcontroladores RISC, cuyo nombre concreto es **AVR**.
Está equipado con catorce **pines de entrada y salida** digitales (E/S) por los que se pueden enviar o leer señales digitales de 0 (valor `LOW` en Arduino) o 5 V (`HIGH`). De estos pines digitales, seis de ellos tienen la capacidad de emitir señales mediante **modulación por ancho de pulsos** (*Pulse-Width Modulation*, o por su acrónimo *PWM* en inglés), por los cuales, se pueden enviar o leer señales de entre 0 y 5 V. También tiene seis pines analógicos de E/S, y es programable con el **entorno de desarrollo integrado** [2] de Arduino mediante un cable USB de tipo B.
Puede alimentarse mediante el propio cable USB cuando se quiera programar, o bien, con una fuente de alimentación externa, ya sea mediante su **adaptador jack** (voltaje recomendado: 12 V), o bien, mediante su pin `Vin` (de 6 a 12 V). También se podría alimentar con 5 V directamente en su pin `5V`, pero las dos formas anteriores son más recomendadas.
Para ser programado (utilizando su *IDE*), se utiliza un lenguaje muy parecido a `C++`, con sus propias bibliotecas; estas pueden ser descargadas desde el propio *IDE*, utilizando el **gestor de librerías**, o bien, desde internet, generalmente desde repositorios de GitHub, donde usuarios de la comunidad de Arduino pueden aportar su grano de arena y poder ser descargadas para uso propio.

Algunas características adicionales del hardware de Arduino UNO R3 son [3]:

- Corriente continua por pin de E/S: 20 mA
- Corriente continua por pines de 3.3 V: 50 mA
- Memoria flash: 32 KB (0.5 KB utilizados por el *bootloader*)
- SRAM: 2 KB
- EEPROM: 1 KB
- Frecuencia de reloj: 16 MHz
- Pin del LED integrado: 13
- Dimensiones: 68.6 x 53.4 mm
- Peso: 25 g

[1] `https://www.microchip.com/en-us/product/atmega328p`
[2] `https://www.arduino.cc/en/software`
[3] `https://store.arduino.cc/products/arduino-uno-rev3`

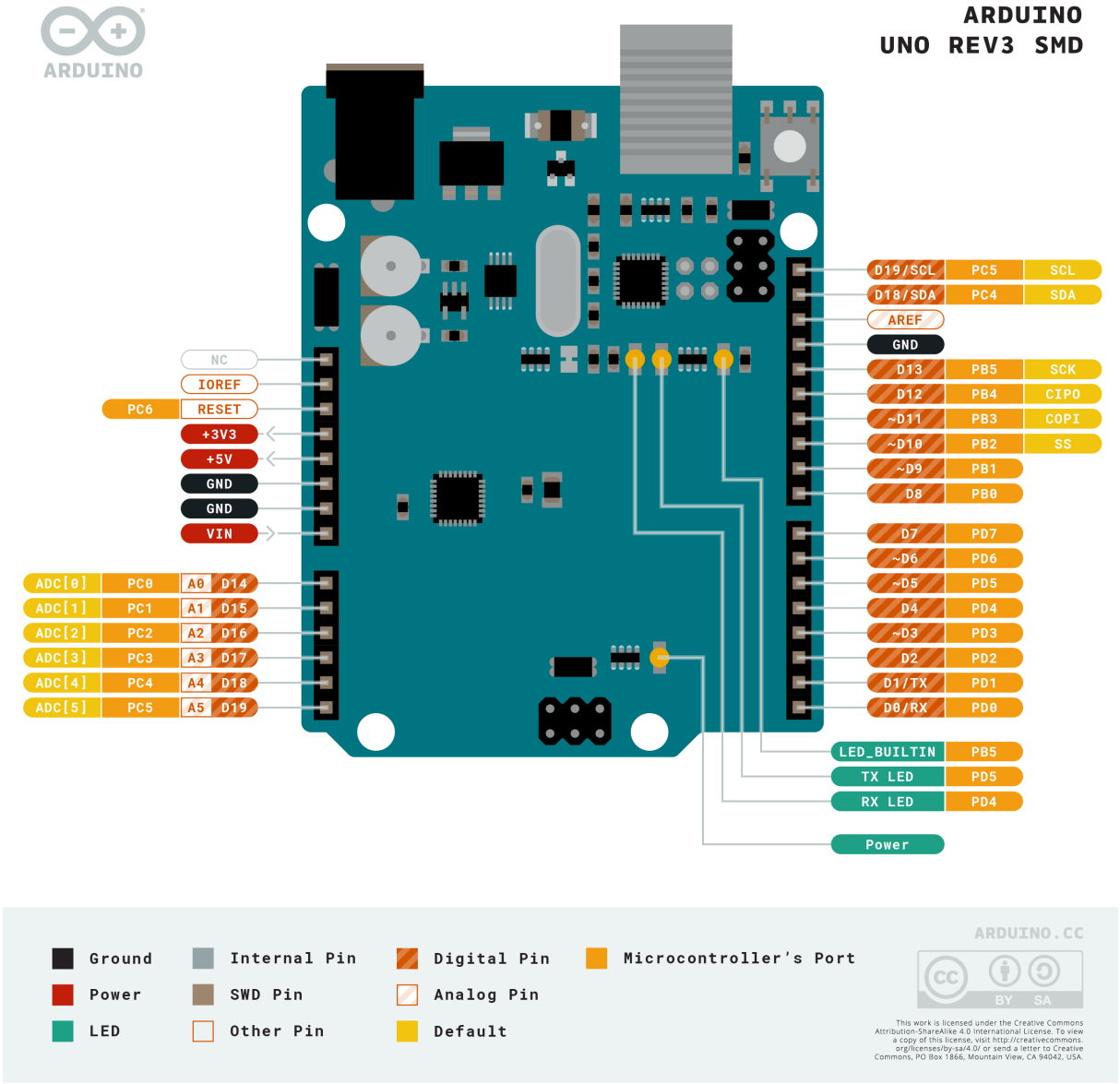

Figura 3.1: Esquema de los pines de Arduino UNO R3.

Como se puede ver en la imagen de arriba [4], los pines digitales se numeran de 0 al 13 y los analógicos, de `A0` a `A5`. Los pines de alimentación son denominados como `3V3`, `5V` y `GND`, que proporcionan una tensión de salida fija, útil para alimentar sensores u otros dispositivos.
También tiene dos *interrupciones* para poder ser utilizadas. Dichas interrupciones no son más que eventos que se ejecutan de forma *invasiva*, es decir, interrumpe la línea de ejecución normal del programa para hacer ejecutar un fragmento de código distinto. En concreto, dichas interrupciones se pueden programar cuando el pin digital 2 (`INT0`) o 3 (`INT1`) cambian de estado de una forma determinada.

[4] `https://docs.arduino.cc/hardware/uno-rev3-smd`

También, es importante destacar los pines que tienen la capacidad de utilizar **modulación por ancho de pulsos**. Concretamente, son los pines digitales 3, 5, 6, 9, 10 y 11; estos tienen la capacidad de simular salidas analógicas con un **ciclo de trabajo** (*duty cycle*) y una **frecuencia constante** determinada.
Esto quiere decir que, para que el Arduino simule estas señales, lo que hará será poner en `HIGH` el valor de salida digital durante un tiempo y después, lo pondrá en `LOW` durante el resto. Y con esto, se vuelve a repetir el mismo procedimiento con el mismo periodo de tiempo.
El voltaje analógico deseado será el **promedio** de la tensión digital a lo largo del tiempo. Por lo tanto, podemos definir el *ciclo de trabajo* como:

- Proporción de tiempo de la señal digital en `HIGH` respecto al tiempo considerado.

Nótese que las señales que genera la modulación por ancho de pulsos son **periódicas**, mientras no varíe la tensión media. Un ejemplo de esto sería el siguiente [5]:

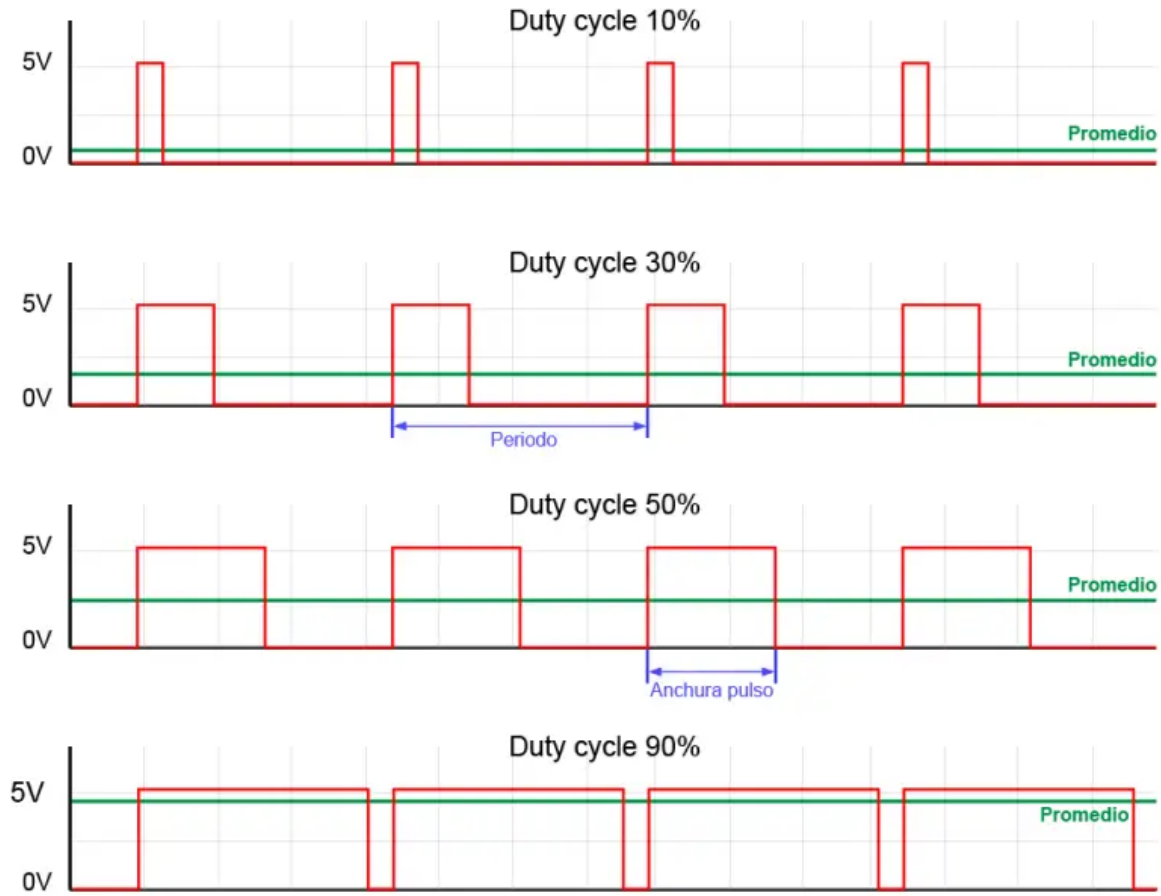

Figura 3.2: Ejemplo de señales generadas por **PWM**.

[5] `https://www.luisllamas.es/wp-content/uploads/2015/09/pwm.webp`

Es evidente que podemos establecer una relación entre la tensión promedio (que es el voltaje «analógico» deseado) y el ciclo de trabajo de la siguiente manera:

$$\overline{V} = (V_{HIGH} - V_{LOW}) \cdot \frac{D.C.}{100}$$

Donde $\overline{V}$ representa el valor «analógico» de salida, V_{HIGH} en nuestro caso, se consideran 5 V y V_{LOW}, 0 V, y $D.C.$ es el ciclo de trabajo [4].
Estos son algunos de los aspectos más importantes de un Arduino, pero hay más características que posee un Arduino, que no serán profundizadas en esta memoria, ya que no son relevantes para el proyecto realizado. Sin embargo, hay algunos pines cuya funcionalidad se explicará más adelante.

3.2. NodeMCU

Es una plataforma de código abierto con **conectividad al internet de las cosas** (*Internet of Things*), la cual, está basada en el `ESP8266`, modelo `ESP-12`, que es un chip compatible con el protocolo TCP/IP, cuyo fabricante es *Espressif Systems* [6]. Se utiliza esta plataforma para suplir una **carencia importante** que tiene el Arduino UNO R3: por sí solo no puede conectarse al *IoT*, ya que no tiene conectividad Wi-Fi, haciéndose imposible enviar datos cómodamente a otros dispositivos, o incluso, poder recibir datos de estos para que el propio Arduino pueda procesarlos y en función de estos, realizar determinadas tareas.
Más específicamente, se puede establecer una **comunicación serial** entre el Arduino y el NodeMCU usando pines de transmisión y recepción de datos (`TX` y `RX`, respectivamente), haciendo que el NodeMCU sea el encargado de enviar y recoger datos del *IoT*, ya sea de otros dispositivos, o de un **bróker MQTT** [5], como veremos más adelante.

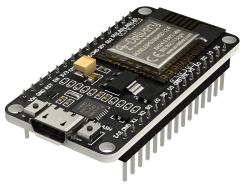

Figura 3.3: Un ejemplar de la plataforma NodeMCU.

Esta plataforma también puede programarse con el entorno de desarrollo integrado del propio Arduino, aunque también se podría hacer con Lua [7] (otro lenguaje de programación), utilizando un MicroUSB para hacerlo. Sus pines también son programables para determinadas tareas, como los del Arduino UNO R3.

[6] `https://www.espressif.com/`
[7] `https://www.lua.org/`

Sin embargo, en este proyecto, solo se ha programado el NodeMCU para recibir datos del Arduino vía comunicación serial de dos pines y enviarlos al *IoT*, o bien enviárselos al Arduino cuando se recibe un mensaje del propio *IoT*.

La distribución de pines del NodeMCU es la siguiente:

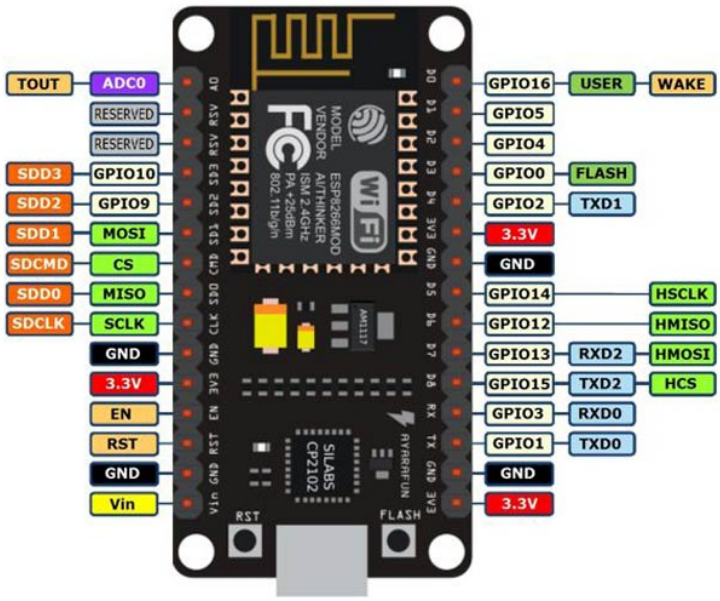

Figura 3.4: Esquema de los pines del NodeMCU.

Se pueden destacar algunas semejanzas con los pines del Arduino UNO R3, como los pines de propósito general (`GPIOx`), el pin de alimentación externa (`Vin`) y el único pin analógico que tiene (`A0`).

Algunas características destacables son:

- Memoria Flash: 4 MB
- Auto-Reset integrado (es útil para no tener que reiniciar manualmente cada vez que se quiera reprogramar el NodeMCU)
- Da soporte al protocolo Wi-Fi 802.11n
- Corriente de salida: 80 mA
- Dimensiones: 49 x 25 mm
- Antena Wi-Fi PCB integrada
- Seguridad WPA/WPA2
- Peso: $\sim$ 19 g

3.3. Raspberry Pi 4B

Es un **ordenador completo** monoplaca (es decir, que tiene todos sus componentes electrónicos, como la **RAM**, los **puertos seriales**, el **microprocesador**... en la placa base) de bajo coste desarrollado en Reino Unido por la *Raspberry Pi Foundation* [8], en asociación con *Broadcom Inc* [9].
Posee una arquitectura de 64 bits, cuyo **sistema en un chip** [10] (*SoC* en inglés) es un `Broadcom BCM2711` y el modelo de su **unidad central de procesamiento** (*CPU*) es un `ARM Cortex-A72`. Posee conectividad a internet gracias a su chip `CYW43455` [11], que proporciona tanto Wi-Fi de doble banda (de 2.4 y 5 GHz) como **Bluetooth 5.0** integrado, y también, de forma alámbrica, gracias a su puerto *Gigabit Ethernet*.
Debido a su capacidad de poder utilizar un sistema operativo [12] ya instalado en una tarjeta **microSD**, puede utilizarse como **ordenador económico**, si es conectado a un monitor y se le instalan periféricos por sus puertos USB, como un teclado y un ratón.
Generalmente, se utiliza alguna de las distribuciones de **Linux**. La más utilizada y recomendada es **Raspberry Pi OS** (antiguamente **Raspbian**), que está basada en Debian y posee herramientas preinstaladas útiles para aprender programación y otras tareas comunes, aunque se podrían instalar más programas mediante su terminal (con el comando `sudo apt-get install [paquete]`). También sería posible instalar otros sistemas operativos, como **Ubuntu**, **Kali Linux**, entre otros [6].
Otras características a destacar de la Raspberry Pi 4B son:

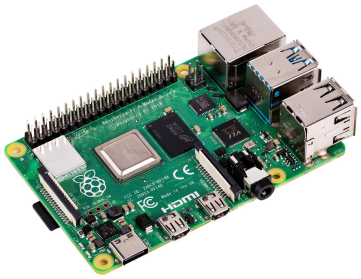

Figura 3.5: Un ejemplar de la Raspberry Pi 4B.

- Memoria RAM: 2, 4 u 8 GB
- Salida de vídeo: Dos puertos micro HDMI con soporte 4K a 60 fps
- Alimentación: USB de tipo C
- GPIO (*General Purpose Input/Ouput*) de 40 pines para conexión a placas de expansión y creación de circuitos electrónicos (como Arduino)
- Almacenamiento: Depende de la microSD utilizada

[8] `https://www.raspberrypi.org/`
[9] `https://www.broadcom.com/`
[10] `https://es.wikipedia.org/wiki/Sistema_en_un_chip`
[11] `https://www.infineon.com/cms/en/product/wireless-connectivity/airoc-wi-fi-plus-bluetooth-combos/wi-fi-5-802.11ac/cyw43455/`
[12] `https://www.raspberrypi.com/software/`

En este caso concreto, se utiliza la Raspberry Pi 4B como un **servidor MQTT**. Esto es posible gracias a un sistema operativo llamado **openHABian**, que es una adaptación a la Raspberry Pi de **openHAB** [13], un software enfocado a la domótica de código abierto escrito en `Java` que permite la automatización y el control de dispositivos inteligentes y sistemas domóticos.
Para que el servidor MQTT funcione, es necesario instalar **Mosquitto** con el terminal de openHABian, que es un software que proporciona todo lo necesario para la creación de **brókeres MQTT**, que se usarán en el proyecto para hacer de intermediario de la transmisión y recepción de datos entre la aplicación móvil desarrollada (se hablará más adelante de ella) y el Arduino, que depende del NodeMCU para la conexión al *IoT*.

3.4. Componentes electrónicos utilizados

Se han utilizado los siguientes sensores y componentes electrónicos para construir el sistema de gestión de peceras, aparte del Arduino y el NodeMCU:

- DS18B20: Es un sensor de temperatura fabricado por **Maxim Integrated** [14]. En este proyecto, se ha utilizado una versión con carcasa estanca para ser sumergido en el agua de la pecera. Posee una interfaz `1-Wire`, que permite transmitir los datos de la temperatura con un solo cable, con un rango de medición de entre -55 y 125 grados Celsius.

Figura 3.6: Sensor DS18B20 utilizado.

- SEN0161: Es un sensor de pH de bajo coste fabricado por **DFRobot** que funciona mediante un conector BNC.
  Ofrece un rango de medición de entre 0 y 14 pH en líquidos, siempre y cuando su temperatura sea de entre 0 y 60 °C. Su ganancia puede ser regulada con el potenciómetro que lleva incorporado en su circuito.
  La página web de su fabricante nos proporciona un código de ejemplo [15] para poder utilizar correctamente este sensor.

[13] `https://www.openhab.org/docs/`
[14] `https://www.maximintegrated.com/en/storefront/storefront.html`
[15] `https://wiki.dfrobot.com/PH_meter_SKU__SEN0161_`

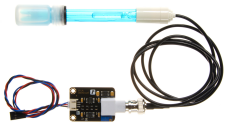

Figura 3.7: Sensor de pH SEN0161.

- HC-SR04: Es un módulo de ultrasonidos fabricado por **OSEPP Electronics** [16] para medir distancias de forma precisa y no invasiva.
  Permite la medición de la distancia entre el sensor y un objeto sin necesidad de contacto físico entre ellos. Funciona gracias a la **emisión de un pulso ultrasónico** desde el transmisor y medir el tiempo que tarda en regresar el eco de dicho pulso al receptor. Utilizando la fórmula de la velocidad del sonido en el aire ($\sim$ 342 m/s) y el tiempo transcurrido, es posible calcular la distancia entre el sensor y el objeto.

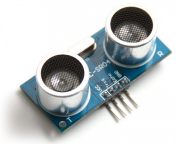

Figura 3.8: Sensor de ultrasonidos HC-SR04.

- DS1307: Es un módulo fabricado **Maxim Integrated** que se utiliza como reloj de tiempo real. En este proyecto, se utiliza para controlar la hora del sistema, ya que el Arduino por sí solo no es capaz de mantener la fecha y hora en el tiempo. Se mantienen estos datos gracias a una **pila alcalina** que tiene incorporada, tras reprogramar el Arduino con la fecha y hora de compilación.

Figura 3.9: Reloj de tiempo real DS1307.

- Calefactor de agua: Es un mecanismo sencillo sumergible que utiliza una **resistencia eléctrica interna** para generar calor a partir de electricidad y distribuirlo uniformemente por todo el agua de la pecera. Se puede apagar o encender mediante un controlador PID para regular la temperatura del agua.

[16] `https://www.osepp.com/`

Figura 3.10: Calefactor utilizado.

- Comedero: Es un aparato electrónico que almacena comida y que tiene un mecanismo que gira cuando sea activado (puede modificarse su circuito interno para que siempre esté activado cuando reciba electricidad, que es el caso).

Figura 3.11: Comedero utilizado.

- Oxigenador de agua: Es un dispositivo que genera burbujas de aire para oxigenar el agua de la pecera.

Figura 3.12: Oxigenador de agua utilizado.

- Bomba de agua: Es un mecanismo para remover el agua de la pecera y ayudar así al calefactor a distribuir el calor de la pecera y al oxigenador a proporcionar más oxígeno al agua con el movimiento generado.

Figura 3.13: Bomba de agua utilizada.

- Tubo de luz fluorescente: Es una lámpara de luz sencilla que se activa siempre y cuando pase corriente eléctrica en ella.

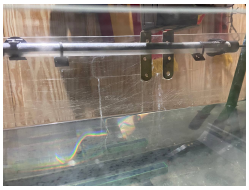

Figura 3.14: Tubo de luz fluorescente utilizado.

- Electroválvula: Es un dispositivo que controla el paso de agua por un **tubo de plástico conectado a un grifo** que siempre está abierto. Permitirá el paso de agua si está encendido y cortará el paso de esta en caso de que esté apagada.

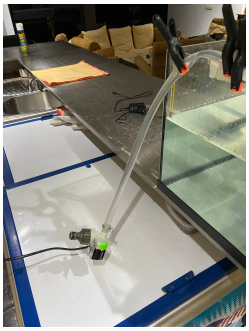

Figura 3.15: Electroválvula utilizada.

También se han utilizado **cables de tipo *jumper*** para unir todo el circuito, un **zumbador** o *buzzer* para crear una alarma que suene cuando el nivel el agua sea demasiado alto o bajo y todo el sistema está montado sobre una **placa de pruebas** o *protoboard*. Por seguridad, para controlar elementos eléctricos que requieran 220 V de tensión, se ha utilizado un **módulo de ocho relés**, controlado por 5 V.
El diagrama electrónico del proyecto se mostrará más adelante.

3.5. El protocolo MQTT

El **protocolo MQTT** (*Message Queuing Telemetry Transport*) es un protocolo de mensajería ligero de **publicación/suscripción** basado en el **modelo cliente-servidor**. Fue diseñado por **IBM** [17] a finales de los años 90 y es altamente utilizado en el ámbito del **internet de las cosas**.

Sus características más destacables son [7]:

- **Ligero**: Optimizado para ser eficiente en entornos con ancho de banda limitado y dispositivos con recursos restringidos, lo que lo hace ideal para aplicaciones *IoT*.
- **Modelo de publicación/suscripción**: Utiliza un patrón de publicar y suscribirse a un *topic*. Los clientes pueden publicar mensajes en *topics*, a los que otros clientes pueden suscribirse para recibir dichos mensajes.
- **Basado en el modelo cliente-servidor**: Los clientes MQTT se conectan a un servidor MQTT (llamado **bróker**), que facilita el intercambio de mensajes entre los clientes.
- **Protocolo asincrónico**: Los mensajes se envían y reciben de manera asincrónica, lo que permite una comunicación eficiente y no bloqueante entre los dispositivos.

Es ampliamente utilizado en entornos de *IoT* para la comunicación entre dispositivos, ya que facilita el intercambio entre sensores, actuadores y otros dispositivos conectados a la red y en aplicaciones donde se requiere un consumo de energía eficiente, debido a su capacidad para mantener conexiones de red de baja latencia y bajo consumo de recursos.

También es posible ser implementado con **medidas de seguridad**, como la autenticación y encriptación, para proteger la seguridad de los datos transmitidos, y también tiene una **alta escalabilidad**, permitiendo a muchos dispositivos conectarse al mismo bróker al mismo tiempo. [18]

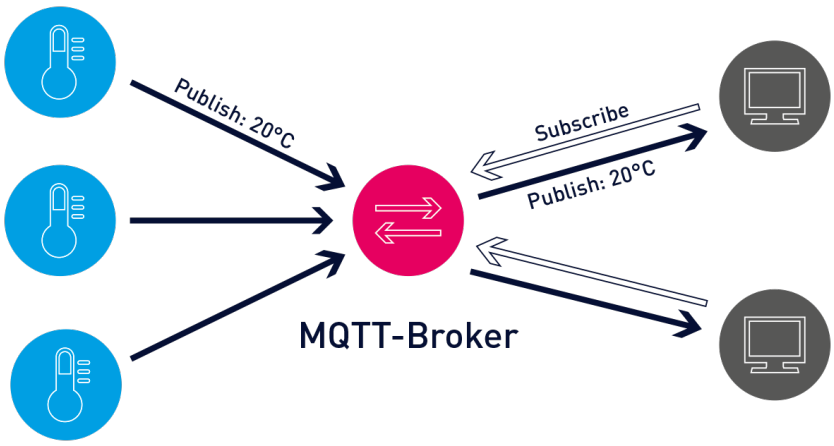

Figura 3.16: Ejemplo ilustrativo de una aplicación que utiliza MQTT.

[17] https://www.ibm.com/es-es
[18] https://hlassets.paessler.com/common/files/infographics/mqtt-architecture.png

3.6. Xamarin Forms

Es un **marco de desarrollo de aplicaciones móviles** que permite a los desarrolladores crear aplicaciones nativas para dispositivos iOS, Android y Windows utilizando un solo código base en `C#`. Es una extensión de la plataforma *Xamarin*, que utiliza `C#` y la plataforma de desarrollo `.NET` para construir aplicaciones multiplataforma [8].

Sus características principales son:

- **Desarrollo multiplataforma**: Con un solo código, es posible utilizar la misma aplicación en diferentes sistemas operativos móviles, como iOS, Android y Windows (Universal Windows Platform).
- **C# como lenguaje principal**: Los desarrolladores pueden utilizar las capacidades de `C#` y el ecosistema de `.NET` para construir aplicaciones. Se utiliza `XAML` para la creación de la interfaz de usuario de cada una de las páginas de la aplicación, siendo compatible usando el mismo código con los sistemas operativos móviles anteriormente mencionados.
- **Interfaz de usuario compartida**: Proporciona una capa de abstracción para la creación de la interfaz de usuario entre las plataformas, lo que simplifica el diseño y la implementación de la IU.
- **Acceso a API nativas**: Permite acceder a las API nativas de cada plataforma a través de Xamarin, lo que posibilita el uso de funcionalidades específicas de cada sistema operativo cuando sea necesario.
- **Herramientas y bibliotecas integradas**: Se integra con **Visual Studio** [19] o **Rider** [20], proporcionando herramientas para el desarrollo, depuración y pruebas de aplicaciones. Además, ofrece acceso a bibliotecas y componentes de `.NET`.
- **Soporte de actualizaciones de plataformas**: Se actualiza regularmente para proporcionar soporte a las últimas actualizaciones y características de las plataformas móviles.

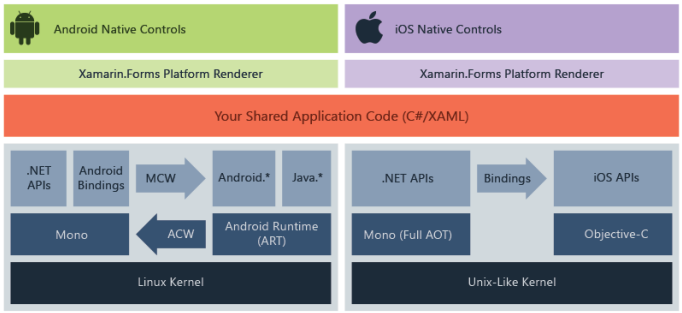

Figura 3.17: Funcionamiento interno de Xamarin Forms

[19] `https://visualstudio.microsoft.com/es/`
[20] `https://www.jetbrains.com/es-es/lp/rider-xamarin/`

3.7. Visual Studio 2022

Es un entorno de **desarrollo integrado desarrollado** por Microsoft. Es una herramienta muy popular entre los desarrolladores de software y se utiliza para crear una variedad de aplicaciones, desde aplicaciones de escritorio hasta aplicaciones web y móviles.

Las principales características de Visual Studio incluyen:

- **Editor de código**: Ofrece un editor de código fuente con resaltado de sintaxis, autocompletado y otras características que facilitan la escritura de código.
- **Depuración**: Proporciona herramientas de depuración poderosas que permiten a los desarrolladores encontrar y corregir errores en su código de manera eficiente.
- **Compilación**: Permite compilar y construir proyectos de software de manera sencilla.
- **Diseñador de interfaces gráficas**: Incluye diseñadores visuales para la creación de interfaces de usuario en aplicaciones de escritorio y móviles.
- **Control de versiones**: Ofrece integración con sistemas de control de versiones, como Git, para facilitar el seguimiento de cambios en el código fuente.
- **Herramientas para desarrollo web**: Incluye herramientas para el desarrollo de aplicaciones web y servicios en la nube.
- **Extensibilidad**:Permite la integración de extensiones y complementos para personalizar el entorno de desarrollo según las necesidades del desarrollador.

4. Metodología de trabajo

Para realizar todo el proyecto de fin de grado, se ha utilizado una **metodología de trabajo ágil**. Estas metodologías consisten en un conjunto de técnicas aplicadas en **ciclos de trabajos cortos**, con el objetivo de que el proceso de entrega de un proyecto sea más eficiente. Hay metodologías de varios tipos, pero la que nos concierne en este trabajo es **Scrum**.
Esta metodología consiste en entregar porciones de un producto al cliente y adaptarse a los posibles cambios que se tenga que hacer a dicho producto. Se basa en principios como la **transparencia**, **inspección** y **adaptación**, con un enfoque en la colaboración y la organización del equipo de trabajo.

4.1. Actores y reuniones

En **Scrum**, los actores pueden tomar diferentes **roles** [9]. En este trabajo, se han llevado a cabo reuniones, donde se determina duración de los ***sprints*** para ir desarrollando poco a poco todo el sistema de control de la pecera y la aplicación móvil. Cada *sprint* puede tener una duración variable, en función de las tareas a desarrollar, llevando un estilo de desarrollo **incremental**. En este caso, los roles se han repartido de la siguiente manera:

- **Product Owner**: Es el encargado de representar las necesidades del cliente, definir las historias de usuario (narrativa corta para expresar las necesidades o requisitos de un sistema desde la perspectiva del usuario final) y gestionar el *backlog* (lista de tareas pendientes, ordenadas por prioridad) del producto. Este rol lo ha *interpretado* el **tutor** del TFG.

- **Scrum Master**: Facilita el proceso Scrum, ayuda a eliminar obstáculos y asegura que el equipo (en este caso, el trabajo es individual) siga los principios y prácticas de Scrum. El **cotutor** ha sido quien ha realizado todo lo mencionado anteriormente, sobre todo con la duración de los *sprints*.

- **Equipo de desarrollo**: Es un equipo multidisciplinario que se encarga de entregar incrementos del producto de alta calidad al final de cada ciclo o *sprint*. El **alumno** ha sido el único miembro del equipo, ya que, como se mencionó anteriormente, el trabajo se ha realizado de forma individual.

En cada *sprint*, se han ido agregando funcionalidades al trabajo y presentados aproximadamente cada tres o cuatro semanas a los tutores del TFG, quienes *interpretaron* los roles anteriormente mencionados. Durante las reuniones, se han discutido posibles mejoras o corregir errores, aparte de redefinir el ***backlog*** para el siguiente ciclo o *sprint*.

4.2. Fases de trabajo

En cada *sprint*, se han llevado a cabo diferentes fases. Estas son las principales tareas que se han realizado, en el siguiente orden:

1. Análisis del problema, requisitos y casos de uso: Discutir cómo diseñar un sistema de gestión de peceras con Arduino que esté controlado por una aplicación móvil y qué herramientas y tecnologías se emplearán para llevar a cabo la construcción de lo anteriormente mencionado, el listado de sus requisitos funcionales y no funcionales, y un diagrama de casos de uso de todas las partes del proyecto. En esta parte, también se incluye la compra de los sensores correspondientes y el NodeMCU.

2. Implementación de un servidor MQTT en una red local: Instalación de **openHABian** en una Raspberry Pi 4B y configuración de un bróker MQTT para recibir y transmitir datos a los clientes MQTT.

3. Creación de una aplicación móvil con conexión a un servidor MQTT: Programar una aplicación móvil en Android que permita visualizar los parámetros actuales de una pecera, establecer ajustes deseados (como la temperatura deseada, hora de encendido de la luz...) y guardar parámetros en la memoria de dicha aplicación.

4. Desarrollo hardware y software del sistema de control: Programar en Arduino un código para controlar la temperatura con un calefactor usando un **controlador PID** en función de la temperatura deseada, controlar el nivel del agua e implementar algunas medidas de seguridad en caso de que el nivel deseado no sea el óptimo, medir el pH del agua y controlar la hora del sistema para encender el comedero diariamente a una cierta hora y encender la luz en un determinado rango de tiempo.

5. Receptor de datos del bróker MQTT: hacer un receptor de datos que se reciban del NodeMCU (este es el encargado de conectarse al **bróker MQTT** y recoger los datos que enviará la aplicación móvil) a través de una comunicación serial UART (*Universal Asynchronous Receiver-Transmitter*), que modificará los **ajustes deseados del sistema**. También, el Arduino enviará al NodeMCU los parámetros medidos de la pecera y este los enviará al bróker MQTT, actualizando así los datos de la pantalla de la aplicación móvil.

4.3. Planificación temporal

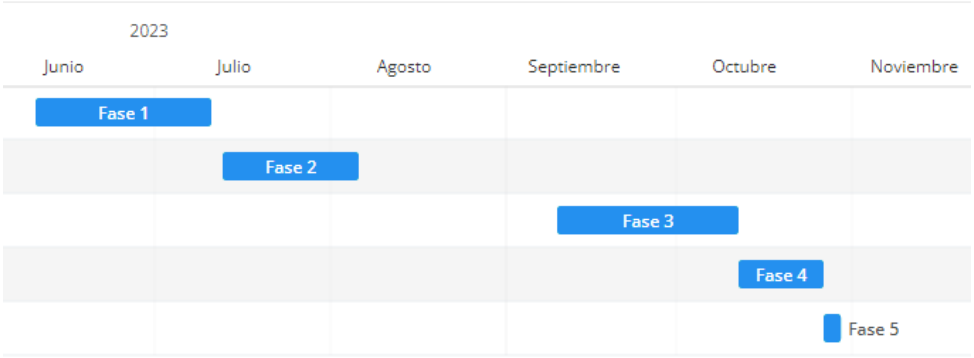

Figura 4.1: Diagrama de Gantt de la duración de cada de una de las cinco fases.

Leyenda del cronograma (las fechas son **aproximadas**):

- Fase 1 → 10/06/2023 - 11/07/2023
- Fase 2 → 13/07/2023 - 06/08/2023
- Fase 3 → 10/9/2023 - 12/10/2023
- Fase 4 → 12/10/2023 - 27/10/2023
- Fase 5 → 27/10/2023 - 30/10/2023

Nótese que entre la segunda y tercera fase ha habido una pausa de un mes porque el alumno decidió aplazar el TFG hasta la convocatoria de diciembre debido a algunas asignaturas que quería recuperar en septiembre del mismo año.

5. Análisis del problema

En este TFG, el problema a resolver es de qué manera se puede hacer **un control casero de una pecera con Arduino** de forma económica e intuitiva. Se desea mostrar en una aplicación móvil el estado actual de una pecera, esto es, mostrar cuatro **parámetros fundamentales** por su página principal: el **pH**, la **hora actual**, **temperatura** y **nivel de profundidad del agua**. También, se quiere controlar ciertos ajustes de la pecera, como la **temperatura y nivel de agua idóneos**, la **hora diaria de encendido del comedero**, las **horas de encendido de la luz** y el **rango de pH permitido del agua**. También sería útil que la aplicación móvil mandase **notificaciones push** al usuario en caso de que se active el comedero, se encienda a luz o algún ajuste **se aleje demasiado de la medida especificada**, ya sea que la temperatura baje o suba un grado de la temperatura ideal, que el nivel de agua demasiado bajo o alto ... Otra característica útil sería poder **guardar datos que aparezcan por pantalla** en la aplicación.
Pero otro problema que surgiría es cómo «conectar» el Arduino con la aplicación para que se puedan comunicar entre ellos, para así poder enviar y recibir datos entre ambos. Y esto sería posible con un servidor ligero, como un **bróker MQTT**: que el NodeMCU que se comunica serialmente con el Arduino (este por sí solo no tiene capacidad de conectarse al internet de las cosas) envíe datos de los sensores a través de *topics* a dicho bróker y la aplicación móvil, que también está conectada al bróker, los reciba suscribiéndose a dichos *topics*. Y la aplicación enviará datos al NodeMCU a través del bróker cuando se quiera cambiar un **ajuste deseado**, este se lo enviará al Arduino y hará los cambios correspondientes.

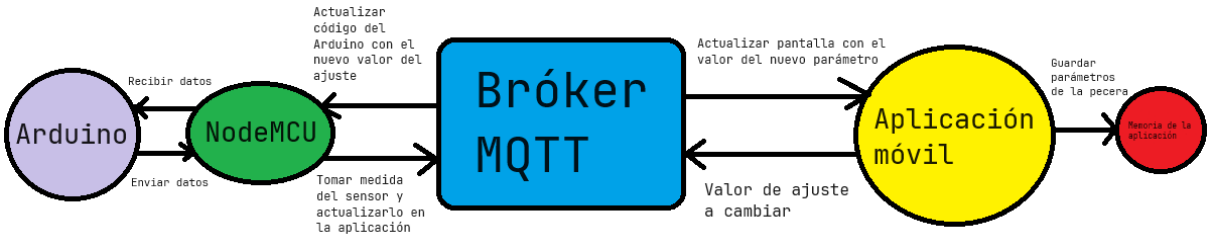

Figura 5.1: Descripción gráfica del análisis del problema.

5.1. Requisitos especificados

Definir los requisitos es **fundamental**, ya que permiten establecer las características y funcionalidades que el proyecto debe cumplir para satisfacer las necesidades del usuario final. Estos requisitos se clasifican en dos tipos: los **requisitos funcionales** y los **requisitos no funcionales**.

5.1.1. Requisitos funcionales

Son aquellos que describen **funciones o tareas** que el sistema debe ser capaz de realizar. Especifican qué debe hacer el sistema, es decir, las acciones que debe llevar a cabo.

Los requisitos funcionales que debe llevar a cabo el proyecto son los siguientes:

Número	Descripción del requisito
1	La aplicación mostrará por pantalla la temperatura, el pH del agua, la hora del sistema y el nivel de profundidad del agua sin interrupciones.
2	La aplicación permitirá guardar los parámetros que se muestran por la pantalla en su memoria.
3	Se podrán editar los ajustes deseados para la pecera con la aplicación. Estos son: **temperatura deseada**, **nivel mínimo y máximo** de agua permitido, **hora diaria de encendido** del comedero, **rango de hora de encendido automático** de la luz y **rango de pH** del agua aceptable.
4	Se podrán ver, editar o borrar los parámetros medidos que estén almacenados en la memoria de la aplicación.
5	El usuario podrá activar y desactivar manualmente la bomba de agua de la pecera, el comedero y la luz cuando lo desee a través de la aplicación.
6	La aplicación móvil enviará una notificación en caso de que algún parámetro anteriormente mencionado se salga del rango del ajuste deseado correspondiente (por ejemplo, que el nivel del agua sea demasiado bajo respecto al nivel especificado de los ajustes).
7	Se podrán desactivar o activar las notificaciones de la aplicación en cualquier momento.
8	Se podrán borrar todas las configuraciones deseadas.
9	El sistema mantendrá automáticamente la **temperatura del acuario** dentro de un rango adecuado de temperatura, determinado por la *temperatura deseada* especificada por el usuario.
10	El sistema mantendrá automáticamente el **nivel de agua del acuario** dentro de un rango adecuado, determinado por los niveles mínimo y máximo especificados por el usuario.
11	El sistema activará de forma automática el **comedero del acuario**, diariamente a la hora especificada por el usuario.
12	El sistema encenderá y apagará, de forma automática, la **luz del acuario**, diariamente a las horas especificadas por el usuario.

Cuadro 5.1: Requisitos funcionales del sistema.

5.1.2. Requisitos no funcionales

Son aquellos que describen **atributos** o **propiedades** del sistema que no están relacionados directamente con las funciones o comportamientos específicos del software. A diferencia de los requisitos funcionales, que se centran en lo que el sistema debe hacer, los requisitos no funcionales se centran en **cómo el sistema debe comportarse** en términos de características no directamente relacionadas con las funciones específicas.

Estos son los requisitos no funcionales del proyecto:

Número	Descripción del requisito
1	El lenguaje de programación del Arduino y el NodeMCU será uno basado en `C++`, con sus correspondientes bibliotecas.
2	El lenguaje de programación de la aplicación móvil será `C#`, usando **Xamarin Forms**.
3	El Arduino y el NodeMCU se comunicarán de forma serial utilizando dos pines para poder enviar y recibir datos entre ellos con formato `JSON`.
4	El NodeMCU se conectará a un **bróker MQTT** para enviar (parámetros de los sensores) y recibir datos (los ajustes deseados o activar un control manual) de la aplicación.
5	El Arduino utilizará los sensores correspondientes para medir los parámetros de la pecera.
6	El bróker MQTT estará almacenado en una Raspberry Pi 4B, usando openHABian como sistema operativo.
7	El control de la temperatura se hará con un calefactor y un controlador PID para mantenerla en una medida determinada por el usuario utilizando la aplicación.
8	Para mantener el agua en movimiento, se utilizará una bomba de agua que podrá ser activada o desactivada manualmente con la aplicación.
9	En el ajuste deseado de *activar alarma con nivel*, sonará una alarma construida con un zumbador en caso de que el nivel sea inferior o igual al especificado.
10	La bomba de agua y el calefactor nunca se encenderán hasta que el nivel sea inferior o igual al nivel de alarma más dos centímetros (por ejemplo, si el nivel de alarma son 5 cm, no se encenderán hasta que haya alcanzado los 7 o menos).
11	Por *nivel de agua* se entiende los centímetros de vacío que hay entre el sensor de distancia (recordemos que se usa un sensor de ultrasonidos HC-SR04) y el propio agua.
12	El vaciado del agua de pecera será manual, mientras que el llenado se hará mediante una electroválvula conectada a un grifo.
13	La memoria de la aplicación donde se guardan los parámetros deseados se hará en una base de datos local. Se usará concretamente **SQLite**.
14	Por simplicidad, la circuitería del gestor de peceras estará instalada sobre una *protoboard*.
15	La aplicación tendrá tres pantallas: una para ver los parámetros actuales de la pecera, otro para programar ajustes deseados y una tabla con los parámetros guardados.
16	Se usará un módulo de ocho relés para controlar elementos electrónicos cuya tensión de alimentación sea de 220 V, como la luz, el comedero o la bomba de agua.
17	La temperatura se mostrará en grados Celsius, el nivel en centímetros y la hora del sistema con el formato `HH:mm:ss` (formato de 24 horas).

18	En caso de que no se especifiquen los ajustes deseados, los valores por defecto serán: ▪ Temperatura deseada: 30 °C ▪ Nivel de alarma (máximo): 5 cm ▪ Hora de encendido del comedero: 00:00:00 ▪ Rango de encendido de la luz 00:00:00 - 00:00:00 ▪ Rango de pH idóneo: 0—1
19	Cada cierto tiempo, se debe calibrar el sensor de pH con su potenciómetro correspondiente.
20	El sensor de nivel de agua, el comedero y la luz estarán instalados en la parte superior de la pecera y se deberá tener cuidado con que no caigan al agua.
21	Opcionalmente, se podrá ver en un minicliente MQTT hecho con `Python` qué datos pasan por el bróker MQTT.
22	Se podrá restablecer la hora del sistema cargando de nuevo el código al Arduino vía USB.
23	La fuente de letra de la aplicación móvil será `JetBrains Mono`.
24	El sistema tratará de regular la medida de los siguientes parámetros en función del ajuste deseado: la **temperatura** y el **nivel del agua**. El pH deberá ser regulado manualmente.
25	Los valores de los parámetros de la pantalla principal de la aplicación se actualizarán cada segundo.
26	La electroválvula se desactivará automáticamente cuando el nivel del agua sea de 2 cm inferior al nivel de alarma y se volverá a activar cuando sea de 6 cm menor al nivel de alarma.
27	El funcionamiento del oxigenador de agua será automático: solo se encenderá si la electroválvula no está encendida, es decir, mientras esté en un nivel óptimo de agua. No hay un control directo sobre él.

Cuadro 5.2: Requisitos no funcionales del sistema.

5.2. Diagrama de casos de uso

Es una **herramienta de modelado** utilizada en el desarrollo de software para visualizar y especificar las interacciones entre diferentes *actores* (ya sean usuarios o sistemas externos) y un sistema de particular. Este tipo de diagrama es parte de la notación `UML` (*Unified Modeling Language*) y se utiliza comúnmente en la fase de análisis para comprender los **requisitos funcionales** de un sistema [10].

Algunos conceptos clave y su papel en el TFG son los siguientes:

- Caso de uso: Representa una **funcionalidad específica** del sistema que puede ser realizada por uno o más actores. Describe cómo un actor interactúa con el sistema para lograr un objetivo particular. Cada caso de uso representa una acción, ya sea en el sistema de la pecera o alguna acción en la aplicación móvil.

- Actor: Representa un **rol o entidad externa** que interactúa con el sistema. Puede ser un usuario humano u otro sistema. Tenemos tres actores: el usuario humano, que controlará la aplicación, la propia aplicación móvil, que enviará avisos y datos al bróker MQTT y el propio sistema, que controlará la pecera.

- Relaciones: Las líneas de asociación entre actores y casos de uso representan las **interacciones**. Un actor que está conectado a un caso de uso indica que el actor participa en la ejecución de ese caso de uso.

- Sistema: El sistema en sí mismo se representa como un rectángulo en el diagrama, y los casos de uso y actores se colocan alrededor de él. En el proyecto tenemos dos *sistemas*, el propio **sistema de gestión de peceras**, y uno que está contenido en él: **la aplicación móvil**, ya que si no existiese, no tendría sentido su existencia.

El propósito del diagrama de casos de uso es **proporcionar una vista de alto nivel de la funcionalidad del sistema** desde la perspectiva del usuario. Es una herramienta efectiva para comunicar los requisitos funcionales y para establecer una base para el diseño y desarrollo posterior del sistema.

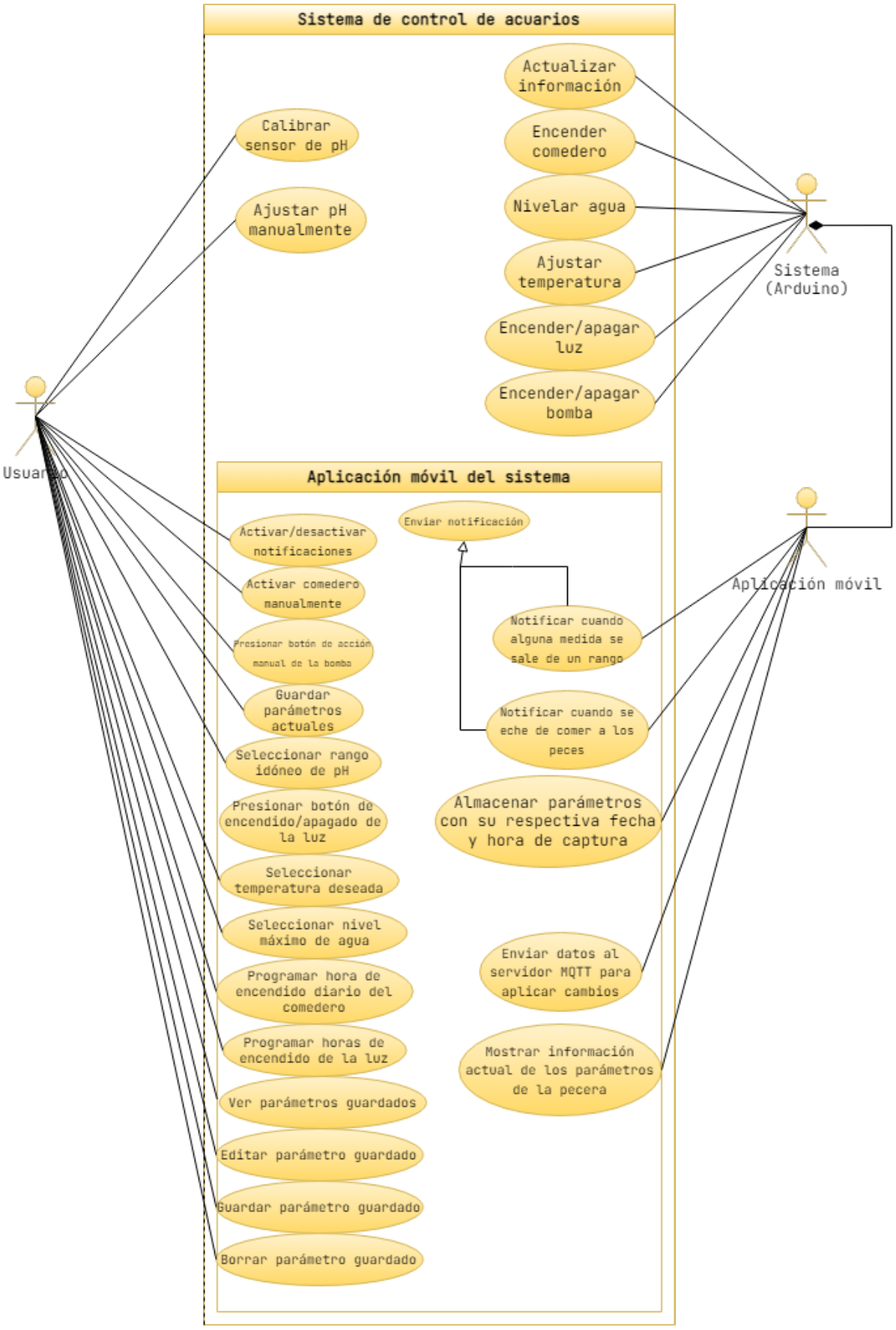

Figura 5.2: Diagrama de casos de uso del sistema completo.

6. Solución propuesta

Este proyecto consta de una arquitectura muy particular, cuyos componentes realizarán un determinado papel clave para el funcionamiento correcto de todo el sistema en general. También se explicarán algunos fragmentos de código importantes para entender mejor la programación y la relación con cada uno de los componentes.

6.1. Diseño y arquitectura

Todo el proyecto está compuesto de cuatro elementos **fundamentales**:

- **La pecera**: Es lo que se quiere controlar de forma automatizada. Se pretende mantener en condiciones óptimas con el fin de reducir el esfuerzo humano que requiere realizar un control manual de esta. Tendrá instalada una serie de actuadores, que son: el comedero, la luz, el oxigenador, la bomba de agua, la electroválvula y el calefactor. Estos serán controlados por el sistema.

- **El sistema de gestión de peceras**: Consta de un **circuito electrónico** con todos los sensores y componentes electrónicos necesarios para regular correctamente el estado de la pecera y mostrar toda la información actualizada de esta al usuario en cualquier momento. Consta también del **Arduino**, que es el que gestiona la pecera, junto con el **NodeMCU**, que es el encargado de recibir la información actual de la pecera (es decir, sus cuatro parámetros fundamentales: pH, nivel de agua, hora y temperatura) del Arduino para enviarla por Wi-Fi al bróker MQTT. También recibirá de este bróker las órdenes de modificar algún ajuste de referencia para enviársela al Arduino por su comunicación serial y que este pueda hacer los cambios necesarios para modificar el estado de la pecera.

- **La Raspberry Pi 4B**: Tiene instalado el bróker MQTT necesario para llevar a cabo las comunicaciones entre la aplicación móvil y el sistema. Este bróker se encarga de establecer una comunicación Wi-Fi entre la aplicación y el propio sistema para que estos puedan interactuar entre sí enviándose órdenes en formato `JSON`, ya sea para actualizar la información del sistema en la aplicación, para activar algún dispositivo manualmente o para actualizar algún ajuste deseado.

- **El dispositivo móvil**: Se ha utilizado un teléfono móvil con **Android** (concretamente, un **Samsung Galaxy S20 FE**) para ejecutar la aplicación móvil. Esta aplicación muestra en tiempo real toda la **información de la pecera** y poder guardarlos para poder ser editados o borrados posteriormente.
  También permite **activar manualmente** tres dispositivos de la pecera: bomba de agua, comedero y la luz fluorescente, **actualizar ajustes deseados** o de referencia. Se conecta al bróker MQTT para enviar los cambios correspondientes realizados en ella y para poder recibir toda la información de la pecera.

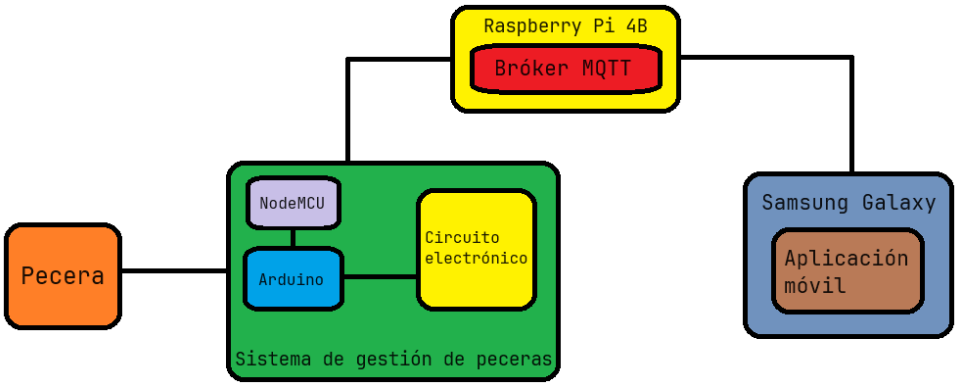

Figura 6.1: Diagrama visual de las relaciones entre los componentes descritos.

6.1.1. Diseño electrónico del sistema de gestión de peceras

El diseño electrónico del sistema de gestión de peceras es el siguiente:

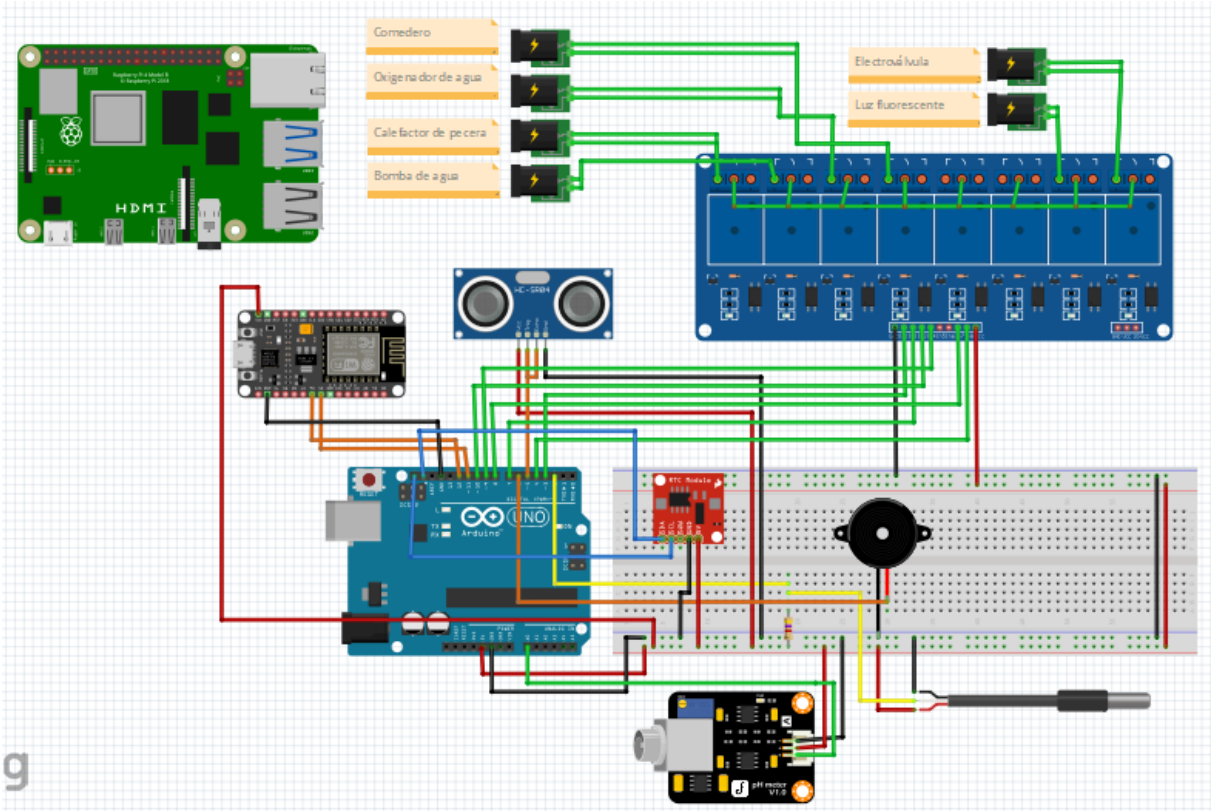

Figura 6.2: Disposición y organización de los componentes utilizados para el proyecto. Programa utilizado para crear el diseño: Fritzing [11]

Se utiliza una *protoboard* por simplicidad, para evitar soldar componentes. Para conectar la mayor parte de la circuitería, se utilizan los pines `5V` y `GND` para conectarlos a los buses del *protoboard*, para así alimentarlos comúnmente.

Vamos a explicar la conexión de los pines del **Arduino UNO R3**:

- Pines 0 (`RX`) y 1 (`TX`): No se utilizan, debido a que se ha utilizado la conexión serial con el ordenador vía USB; se podrían utilizar, pero no se podría acceder al serial del ordenador para visualizar datos por pantalla por su puerto serial.
- Pin 2 (`INT0`): Se utiliza para **recibir la temperatura** del sensor DS18B20. Nótese que está conectado también a una resistencia *pull-up* de 4700 Ω, para poder recibir los datos (con protocolo `1-Wire`) de forma correcta.
- Pin 3 (`INT1`): Se utiliza para controlar el **calefactor** de la pecera. Se apaga y se enciende en función del valor del controlador PID programado en el código del proyecto para mantener una temperatura deseada.
- Pin 4: Controla la **electroválvula** para llenar la pecera. Se apagará o se encenderá en función del nivel de agua deseado.
- Pin 5: Se utiliza para controlar los pines `TRIG` y `ECHO` del **sensor de ultrasonidos**. Gracias a una biblioteca que existe (`NewPing.h`), podemos conectar con un solo cable estos dos pines, ahorrándonos así un pin en el Arduino que se puede utilizar para conectar otro dispositivo.
- Pin 6: Se utiliza para controlar la **alarma** del sistema. Se encenderá en caso de que el nivel de agua alcance el nivel de alarma, o si está demasiado bajo.
- Pin 7: Se utiliza para controlar la **bomba de agua** de la pecera. Se puede apagar y encender con la aplicación móvil realizada.
- Pin 8: Enciende y apaga la **luz fluorescente** de la pecera. Puede encenderse manualmente o programando un rango de hora de encendido diario con la aplicación.
- Pin 9: Enciende el **comedero**. De forma análoga a la luz, también se puede programar cuándo se encenderá el comedero o hacerlo manualmente con la aplicación.
- Pin 10: Enciende o apaga el **oxigenador del agua**. Esta solo se mantendrá encendida si el nivel de agua se mantiene en un rango aceptable.
- Pines 11 y 12: Son los pines **transmisor** y **receptor** de datos del Arduino. Está conectado al NodeMCU serialmente para enviar y recibir datos de ella (ya que es dependiente de esta plataforma para conectarse al IoT).
- Pines `SDA` y `SCL`: Utilizados por el **RTR DS1307** para actualizar la hora del sistema. Se utiliza el protocolo **I2C** [12] para ello.
- Pin analógico 0 (`A0`): Se utiliza para **recibir datos** del sensor de pH **SEN0161**. Envía datos analógicos de forma continua y luego se hace una media aritmética para determinar el pH del agua.

La **Raspberry Pi 4B** no tiene conexión alguna con el circuito, ya que solo se encarga de mantener activo el bróker MQTT para enviar y recibir datos tanto del NodeMCU como de la aplicación móvil diseñada.

Para conectar los dispositivos cuya tensión de alimentación sea de 220 V (calefactor, luz fluorescente, comedero, oxigenador de agua, bomba de agua y electroválvula), se utiliza un **módulo de ocho relés** alimentado con 5 V para controlar el encendido y apagado de estos, con los pines del Arduino correspondientes.

6.2. Transmisión de datos usando MQTT

Vamos a explicar la comunicación entre la aplicación móvil y el Arduino utilizando este protocolo.

6.2.1. Servidor MQTT

Se utiliza una Raspberry Pi 4B con sistema operativo openHABian para mantener el bróker MQTT conectado. Funciona mediante una red Wi-Fi local y es necesario instalar **Mosquitto** en él para poder crearlo satisfactoriamente. Opcionalmente, se podrá configurar el protocolo TLS/SSL para una mayor seguridad y el nombre del host a utilizar.

Podemos conectarnos vía SSH con **Windows PowerShell**, **Bitvise SSH Client** [21] o **PuTTY** [22] al terminal de la Raspberry Pi 4B para poder configurarla o probar el bróker MQTT.

Figura 6.3: Pantalla de bienvenida tras conectarse vía SSH a openHABian.

[21] https://www.bitvise.com/ssh-client-download
[22] https://www.putty.org/

Algunos comandos interesantes son:

- `mosquitto_sub -h [nombre del host] -t [nombre del topic]`: Se suscribe al *topic* especificado en la bandera -t del bróker especificado en la bandera `-h`. Puede utilizarse para hacer pruebas.
- `mosquitto_pub -h [nombre del host] -t [nombre del topic] -m [mensaje a publicar]`: Envía al *topic* especificado en la bandera `-t` del bróker especificado en la bandera `-h` el mensaje o *payload* de la bandera `-m`. Si se quiere poner un mensaje con espacios, se deben utilizar comillas dobles (“ ”).

Veamos una prueba de estos dos comandos:

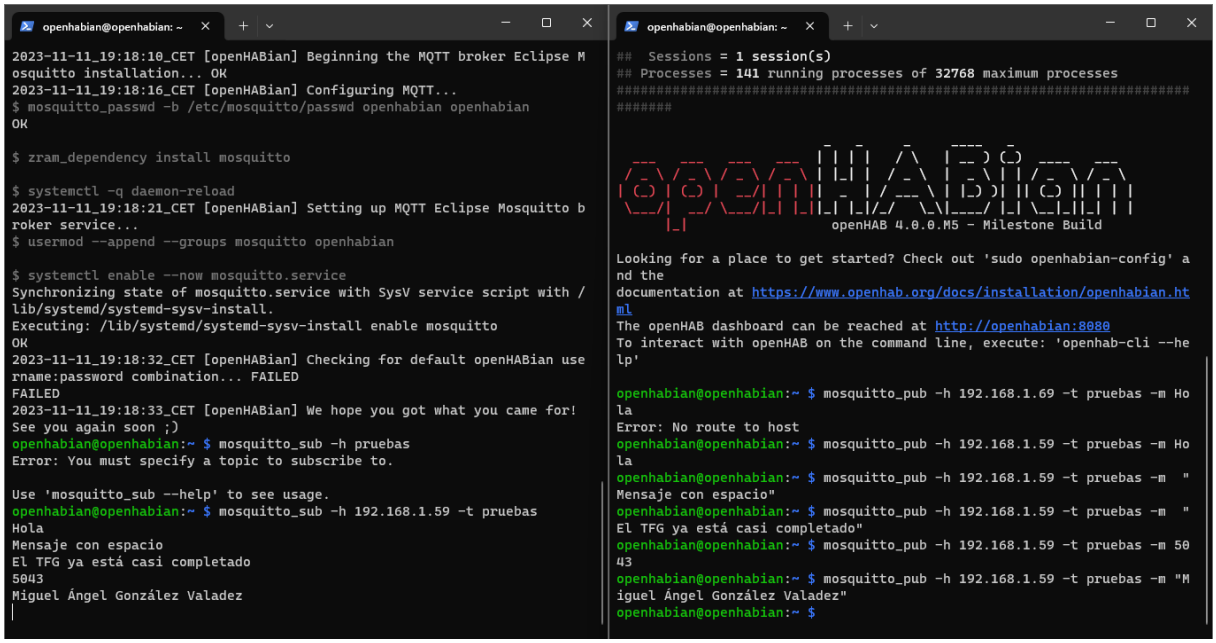

Figura 6.4: Prueba de los comandos MQTT.

Nótese que el nombre de nuestro bróker es **192.168.1.59** y se han abierto dos instancias de **Windows PowerShell**, una para suscribirse a un *topic* de prueba y otro para enviar mensajes.
Podemos ver si la Raspberry Pi 4B tiene Mosquitto activado con el comando `sudo systemctl status mosquitto`.

6.2.2. Cliente MQTT

Son los dispositivos que se conectan al bróker MQTT explicado en la subsección anterior.
En nuestro proyecto, hay dos clientes principales: la **aplicación móvil** y el **NodeMCU**.

- La **aplicación móvil** tratará de conectarse al bróker MQTT nada más iniciarse. Cuando la conexión sea exitosa, se podrán órdenes, como apagar y encender la bomba de agua, el comedero o la luz de forma manual, y establecer ajustes deseados.

- El **NodeMCU** también se conectará al bróker MQTT para recibir las órdenes de la aplicación móvil y para actualizar la pantalla principal de esta con los datos de los sensores.

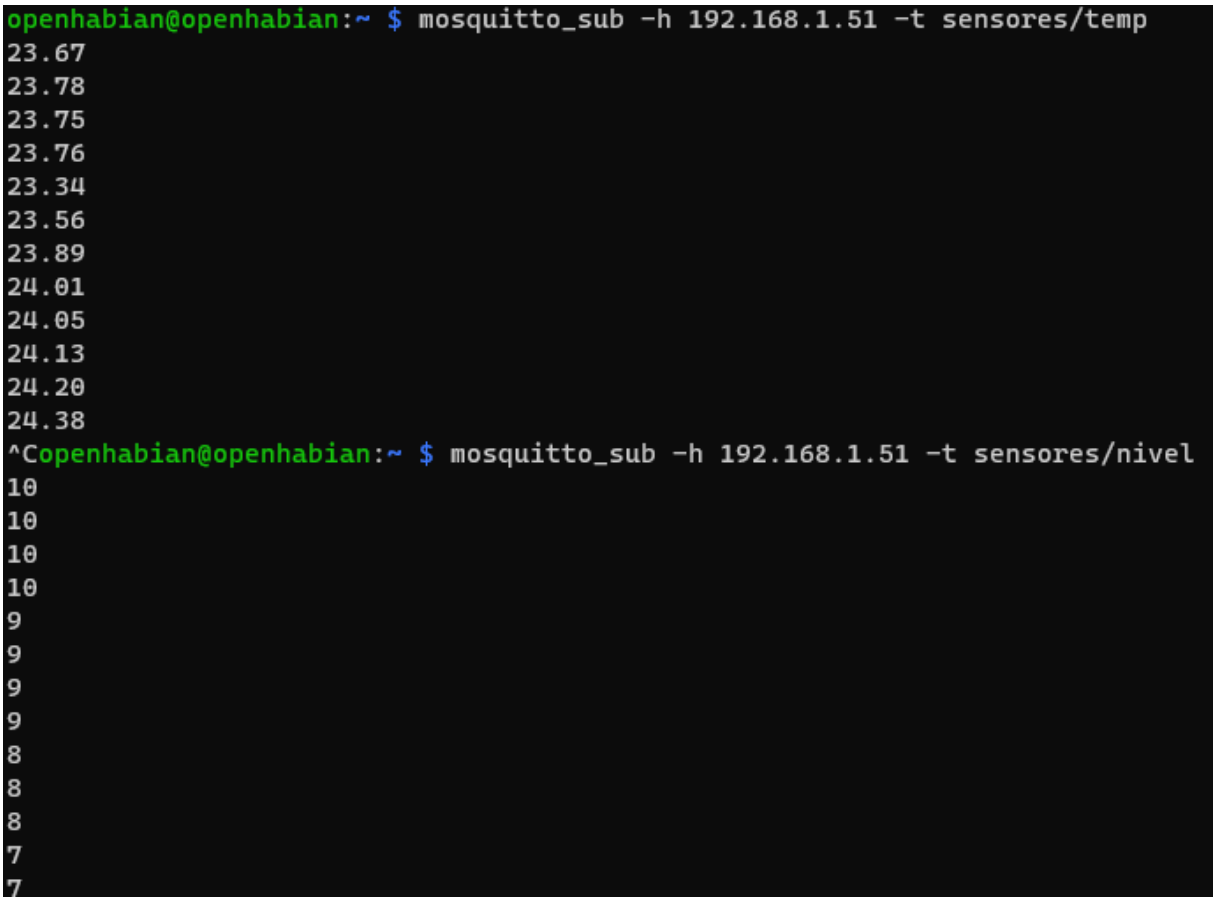

Figura 6.5: Los sensores de temperatura y de nivel de agua.

Como se puede ver en la imagen de arriba, el NodeMCU está enviando al bróker MQTT datos de los sensores del Arduino de forma continuada. Todos los *topics* utilizados para la comunicación entre el NodeMCU y la aplicación móvil son los siguientes:

- `avisos`: Lo utilizan tanto el NodeMCU como la aplicación móvil para dar un mensaje de bienvenida con un UUID (versión 4) aleatorio.
- `sensores/temp`: Lo utiliza el sensor de temperatura DS18B20 para enviar constantemente el valor de la temperatura actual a la aplicación móvil.
- `sensores/RTR`: Lo utiliza el reloj de tiempo real DS1307 para enviar la hora del sistema a la aplicación móvil.
- `sensores/pH`: Lo utiliza el sensor de pH SEN0161 para enviar el valor del pH a la aplicación móvil.
- `sensores/nivel`: Lo utiliza el sensor de ultrasonidos HC-SR04 para enviar el valor del nivel de agua a la aplicación móvil.

- `ajustes/controlManual`: Lo utiliza la aplicación móvil para enviar mensajes de encendido o apagado manual de la bomba de agua, comedero o luz fluorescente al NodeMCU.
- `ajustes/tempDeseada`: Lo utiliza la aplicación móvil para enviar la temperatura deseada al NodeMCU.
- `ajustes/horaComedero`: Lo utiliza la aplicación móvil para enviar la hora de encendido diaria del comedero al NodeMCU.
- `ajustes/rangoLuz`: Lo utiliza la aplicación móvil para enviar el intervalo de encendido de la luz fluorescente diario al NodeMCU.
- `ajustes/nivelAlarma`: Lo utiliza la aplicación móvil para enviar el nivel de agua al que se activará la alarma al NodeMCU.
- `ajustes/rangoPH`: Lo utiliza la aplicación móvil para enviar el rango de pH idóneo al NodeMCU.

Adicionalmente, se ha hecho un **minicliente MQTT** con **Python** para poder ver con todo lujo de detalle todos los mensajes enviados a todos los *topics* tanto por el NodeMCU como por la aplicación móvil.

Figura 6.6: El minicliente Python funcionando.

6.3. Aplicación móvil utilizada

La **aplicación móvil** es sencilla y consta de tres páginas. Cada una, tiene una funcionalidad distinta.
Antes de cargar la aplicación, tratará de conectarse al bróker MQTT para que sea completamente funcional. Veamos el siguiente código:

```
1  private async void CreateMQTTClient()
2  {
3      var dbPath = Path.Combine(Environment.GetFolderPath(
4          Environment.SpecialFolder.LocalApplicationData), "medidas.db3");
5      Connection = new DatabaseConnection(dbPath);
6
7      if (await Connection.Table<ConfiguracionActual>().CountAsync() == 0)
8          await Connection.InsertAsync(new ConfiguracionActual());
9
10     try
11     {
12         var options = new MqttClientOptionsBuilder()
13             .WithTcpServer(broker, 1883)
14             .WithClientId(clientId)
15             .WithCleanSession()
16             .Build();
17
18         var connectResult = await MqttClient.ConnectAsync(options);
19         if (connectResult.ResultCode ==
               MqttClientConnectResultCode.Success)
20         {
21             await MqttClient.SubscribeAsync("sensores/temp");
22             await MqttClient.SubscribeAsync("sensores/pH");
23             await MqttClient.SubscribeAsync("sensores/RTR");
24             await MqttClient.SubscribeAsync("sensores/nivel");
25             await MqttClient.SubscribeAsync("ajustes/controlManual");
26             var message = new MqttApplicationMessageBuilder()
27                 .WithTopic("avisos")
28                 .WithPayload($"¡Conectado! ID del cliente: {clientId}")
29                 .WithQualityOfServiceLevel(MqttQualityOfServiceLevel.
30                     AtMostOnce)
31                 .WithRetainFlag(false)
32                 .Build();
33             await MqttClient.PublishAsync(message);
34         }
35     }
36     catch (MqttCommunicationException)
37     {
38         await Shell.Current.DisplayAlert("Error en la conexión al
               servidor",
39                 $"Error al establecer una conexión con la dirección
                       {broker}.",
40                 "Aceptar");
41     }
42 }
```

Como se puede ver en el código anterior, se está utilizando la biblioteca `MQTTnet` [23], que sirve para crear conexiones a servidores MQTT o hasta para crear un servidor MQTT propio. Lo más relevante de esta sección de código es cómo intenta crear una conexión al bróker MQTT y luego, si lo ha hecho correctamente, se suscribirá a los *topics* de los sensores de Arduino. En caso de que falle la conexión, lanzará una alerta al usuario indicando que ha fallado la conexión. Aunque la aplicación se podrá seguir utilizando, será prácticamente inservible, ya que no podrá enviar y ni recibir datos. Este código está localizado en el fichero `App.xaml.cs` del proyecto.

6.3.1. Ver parámetros actuales, control manual y guardar parámetros

Esta es la **pantalla principal** de la aplicación. Se pueden ver cuatro celdas, cada una con uno de los **cuatro parámetros principales** de la pecera. Debajo, tenemos el control manual de la pecera. Podemos encender y apagar manualmente la bomba, el comedero y la luz con cada uno de los conmutadores, que representan el estado actual (encendido o apagado) en la pecera. También se puede ver un **botón** que dice **GUARDAR MEDIDAS ACTUALES**, que guardará en la memoria de la aplicación los cuatro parámetros que aparezcan por pantalla.

Figura 6.7: Pantalla principal de la aplicación.

[23] `https://github.com/dotnet/MQTTnet`

6.3.2. Establecer ajustes de la pecera

Es la pantalla donde se podrán enviar las **medidas o ajustes deseados**. Se pueden ver las siguientes seis celdas, cada una con un *checkbox*, que servirá para mandar al servidor MQTT los cambios a realizar:

- Temperatura deseada: La temperatura deseada que se desea en la pecera en grados Celsius.
  Está limitada entre los 20 y 32 °C.
- Activar alarma con nivel: El nivel de agua deseado en centímetros.
  Está limitado entre los 2 y 26 cm.
- Encender el comedero a: La hora del encendido diario del comedero.
- Encender luz durante: El rango de horas en el que estará encendido la luz fluorescente.
- Rango de pH deseado: El rango de pH deseado del agua.
  El límite inferior no puede ser mayor al superior ni tampoco pueden ser negativos ambos límites.
- Activar notificaciones: Aquí se decide si activar o desactivar las notificaciones push de la aplicación.

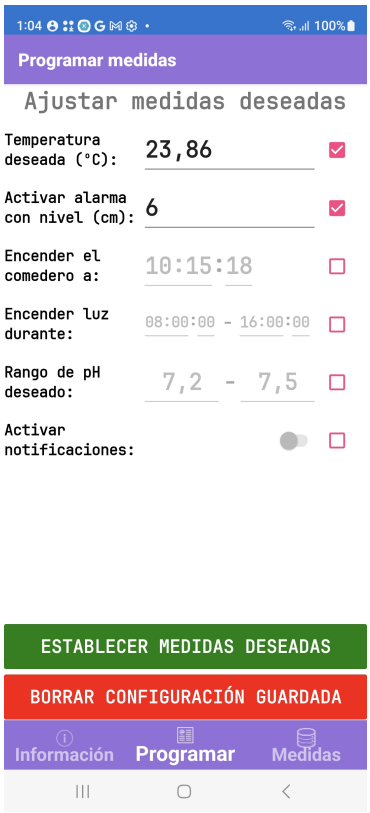

Figura 6.8: Pantalla de ajustes deseados.

Debajo de las celdas, se pueden ver dos botones, uno que dice **ESTABLECER MEDIDAS DESEADAS**, que guardará los cambios, los enviará al NodeMCU, y luego al Arduino para que el código de este cambie y se apliquen correctamente los ajustes deseados, y el otro, que dice **BORRAR CONFIGURACIÓN GUARDADA**, que borrará los ajustes deseados guardados y pondrá todos los ajustes por defecto, tanto en la aplicación como en el Arduino.

6.3.3. Editar parámetros guardados

En esta pantalla se pueden visualizar todas las medidas que el usuario haya guardado con el botón **GUARDAR MEDIDAS ACTUALES** en la primera pantalla. Incluye la información completa de lo que se guardó. Debajo del todo, se puede apreciar un botón que dice **BORRAR MEDIDAS GUARDADAS**. Esto borrará todas las celdas de la lista de parámetros guardados.

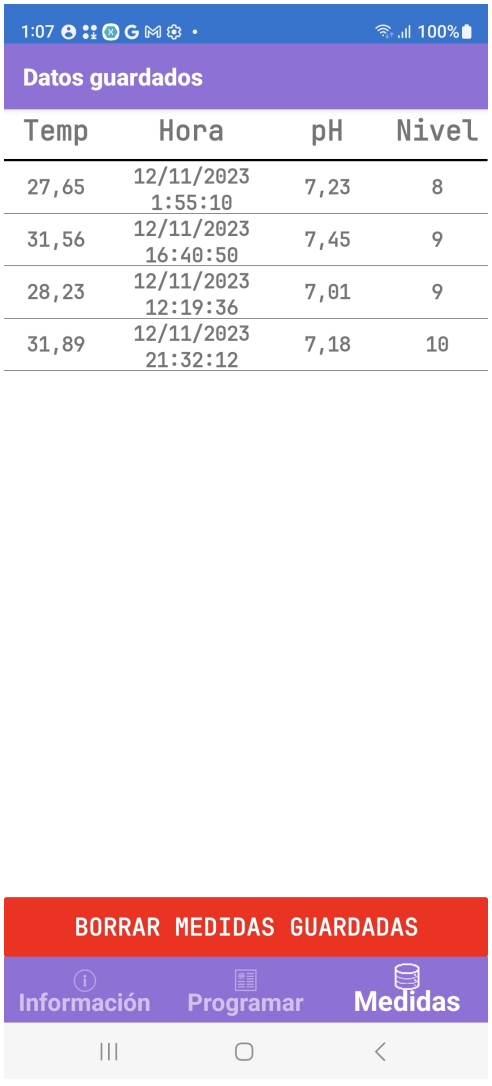

Figura 6.9: Pantalla de visualización de parámetros guardados.

Si se toca cualquier celda de la lista, redirigirá a una página, donde el usuario podrá editar a su conveniencia cualquiera de los cuatro parámetros que desee. Debajo de esta página hay tres botones:

- **CANCELAR**: Regresa a la lista de parámetros guardados sin guardar las modificaciones hechas.
- **GUARDAR CAMBIOS**: Guarda los cambios realizados y regresa a la página anterior.
- **BORRAR MEDIDA**: Borra de la lista la celda elegida.

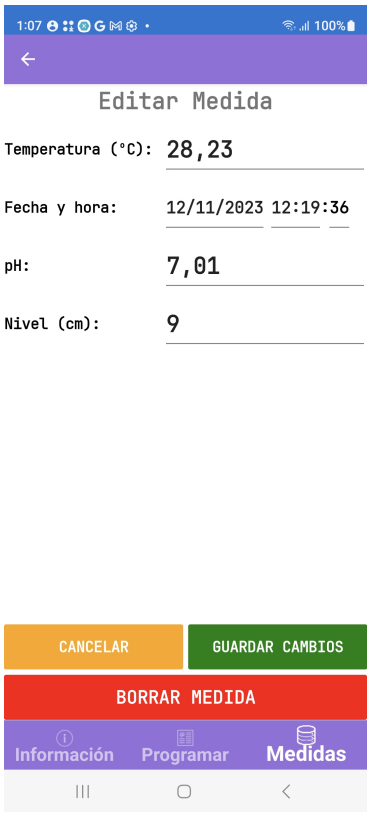

Figura 6.10: Pantalla de edición de una celda.

6.3.4. Memoria interna de la aplicación

La persistencia tanto de los **parámetros guardados** como de los **ajustes deseados** es posible gracias a una base de datos embebida, **SQLite**. Nada más empezar la aplicación, se creará un fichero interno dentro de la memoria del móvil llamado `medidas.db3`. Esto creará dos tablas internas: `Medida` y `ConfiguracionActual`. La primera almacenará una tabla de elementos con el siguiente modelo:

```
[Table("Medida")]
public class Medida
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    public double Temperatura { get; set; }
    public DateTime FechaYHora { get; set; }
    public double PH { get; set; }
    public int Nivel { get; set; }

    public override string ToString()
    {
        return $"({Id}, {Temperatura}, {FechaYHora}, {PH}, {Nivel})";
    }
}
```

Este modelo representa una tupla de cuatro elementos, los elementos principales más el identificador, que se autoincrementará con cada inserción.
Y la segunda será una tabla de un **único elemento** que simplemente almacenará los ajustes deseados de la pecera. Este es el código del modelo:

```
[Table("ConfiguracionActual")]
public class ConfiguracionActual
{
    [PrimaryKey, AutoIncrement]
    public int Id { get; set; }
    public double? TemperaturaDeseada { get; set; }
    public int? NivelAlarma { get; set; }
    public TimeSpan? HoraComedero { get; set; }

    public TimeSpan? HoraLuzInferior { get; set; }
    public TimeSpan? HoraLuzSuperior { get; set; }

    public double? PHMin { get; set; }
    public double? PHMax { get; set; }
    public bool Notificaciones { get; set; }

    public override string ToString()
    {
        return $"({Id}, {(TemperaturaDeseada.HasValue ?
            (TemperaturaDeseada + " C") : "X")}, " +
            $"{(NivelAlarma.HasValue ? NivelAlarma + " cm" : "X")}, " +
            $"{(HoraComedero.HasValue ? ${HoraComedero}" : "X")}, " +
            $"{(HoraLuzInferior.HasValue ? ${HoraLuzInferior} -
                {HoraLuzSuperior}" : "X")}, " +
            $"{(PHMin.HasValue ? PHMin + " - " + PHMax : "X")}, " +
            $"{(Notificaciones ? "Con " : "Sin ")} notificaciones)";
    }

}
```

Se puede apreciar que casi todos los atributos son `nullable`, es decir, puede haber un valor almacenado o no. Los métodos `ToString()` pueden ser utilizados para depurar.

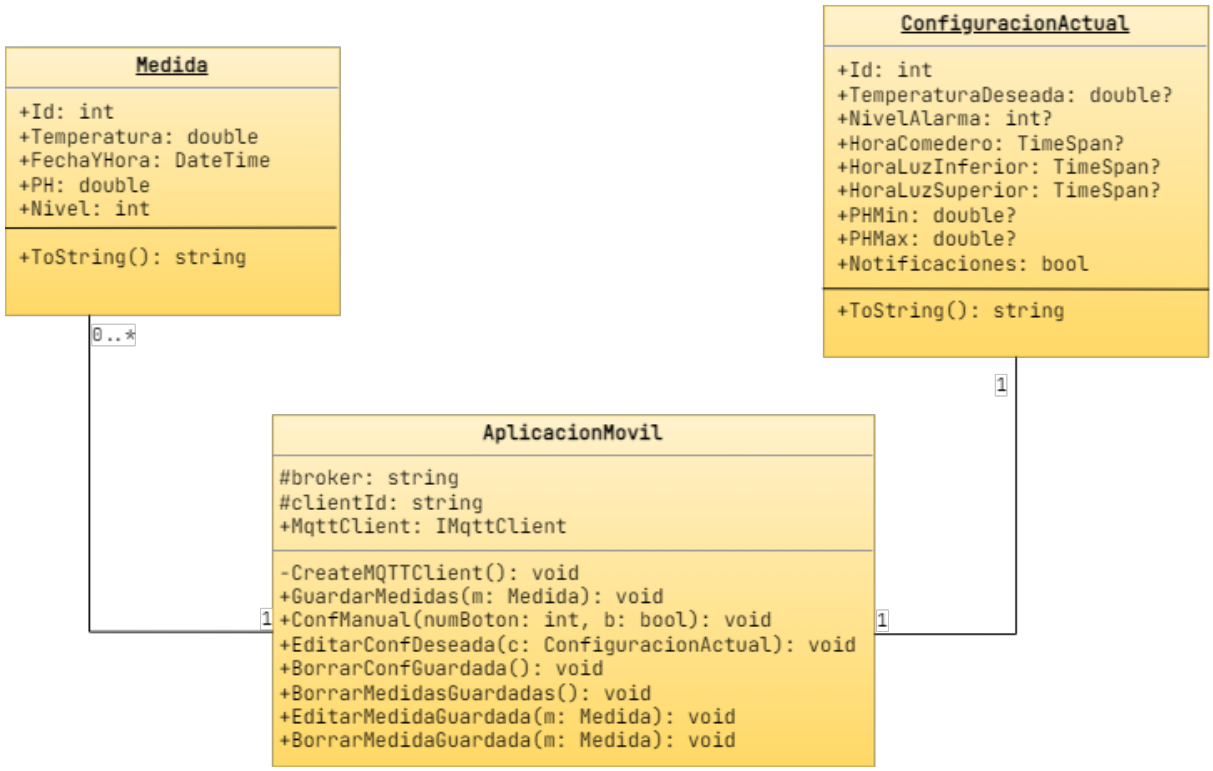

Figura 6.11: Diagrama de clases de la memoria de la aplicación.

7. Implementación y pruebas

En esta sección, se va a explicar el código implementado del sistema de gestión de peceras y algunas pruebas documentadas para verificar su comportamiento.

7.1. Código del Arduino UNO R3

El código del Arduino UNO R3 está compuesto por cinco ficheros. Cada uno tiene una funcionalidad distinta:

7.1.1. Medición de la temperatura y uso del controlador PID

```
#include <OneWire.h>
#include <DallasTemperature.h>
#include <PID_v1_bc.h>

double temp, valor_PID;                                    // Valores para guardar la última temperatura tomada y el valor del PID.
const double Kp = 20, Ki = 0.91, Kd = 1.7;                 // Constantes del controlador PID.
const unsigned long MEDICION = 1000, CONTROL = 5000;       // Periodo de medición.
unsigned long t_anterior_temp, t_anterior_control;         // Tiempo actual del programa y de última medición, respectivamente.
bool activarSubciclo = false;                              // Activar subciclo de trabajo del contrador PID.

OneWire one_wire(pin_temp);
DallasTemperature sensor(&one_wire);
PID myPID(&temp, &valor_PID, &temperatura_deseada, Kp, Ki, Kd, DIRECT);

void controladorPID_setup() {
  sensor.begin();
  myPID.SetMode(AUTOMATIC);
  myPID.SetSampleTime(5000);
  pinMode(pin_calefactor, OUTPUT);
  digitalWrite(pin_calefactor, OFF);
}

void tomar_temperatura() {
  sensor.requestTemperatures();
  temp = sensor.getTempCByIndex(0);
}

void controladorPID_loop() {
  apagar_comedero();
  unsigned long t_actual = millis();
  if (t_actual - t_anterior_temp >= MEDICION) {
    tomar_temperatura();
    enviarJSON("sensores/temp", String(temp, 2));
    t_anterior_temp = t_actual;
  }

```

```
37    if (seguridad_calefaccion) {
38      if (t_actual - t_anterior_control >= CONTROL && myPID.Compute()) {
39        activarSubciclo = 0 < valor_PID && valor_PID < 20.0;
40        digitalWrite(pin_calefactor, valor_PID > 0 ? ON : OFF);
41        t_anterior_control = t_actual;
42      }
43    } else {
44      digitalWrite(pin_calefactor, OFF);
45    }
46
47    if (activarSubciclo && t_actual - t_anterior_control >= valor_PID *
          CONTROL / 20.0) {
48      activarSubciclo = false;
49      digitalWrite(pin_calefactor, OFF);
50    }
51  }
```

Este código se encuentra en el fichero `ControladorPID.ino`, y se pueden destacar varias cosas:

1. Para realizar el controlador PID, se ha hecho uso de una biblioteca externa, `PID_v1_bc.h` [24]. Esta tiene una clase llamada `PID`, que tomará los siguientes parámetros:
   - `double* temp`: El valor actual de la temperatura.
   - `double* valor_PID`: Una variable para sobreescribir el valor devuelto de la última salida del controlador PID.
   - `double* temperatura_deseada`: La temperatura deseada. Esta puede cambiar en el tiempo.
   - `double Kp`: La parte proporcional del controlador.
   - `double Ki`: La parte integral del controlador.
   - `double Kd`: La parte derivativa del controlador.
   - `int ControllerDirection`: Es el modo de operación del controlador PID. Se utiliza la constante `DIRECT`, ya que se desea el valor de salida del PID se incremente cuando la temperatura real es más baja que la deseada.
2. Se utilizan las bibliotecas `OneWire.h` [25] y `DallasTemperature.h` [26] para crear una interfaz *1-Wire* y utilizarla para medir temperaturas con el sensor de temperatura DS18B20.
3. Cada segundo, se actualiza el valor de la temperatura y se envía al NodeMCU para actualizar la pantalla principal de la aplicación móvil (utilizando la función `enviarJSON`).
4. La variable booleana `seguridad_calefaccion` mantiene apagado el calefactor mientras el nivel de agua no sea óptimo. De ser así, cada cinco segundos se actualiza el valor de salida del PID. Si este es **mayor que cero**, se encenderá el calefactor. En caso contrario, se apagará.

[24] `https://www.arduino.cc/reference/en/libraries/pid_v1_bc/`
[25] `https://www.arduino.cc/reference/en/libraries/onewire/`
[26] `https://www.arduino.cc/reference/en/libraries/dallastemperature/`

5. Las constantes del controlador PID (`Kp`, `Ki` y `Kd`) pueden modificarse con el fin de mejorar el tiempo de respuesta del calefactor.

6. Para actualizar el valor del controlador PID, se hace usado de una función llamada `Compute`, que devolverá un valor función en función de si se ha podido actualizar el valor del PID.

7. Para evitar que el controlador esté encendido o apagado durante los cinco segundos del periodo de cálculo del valor del PID, se utiliza un subciclo de trabajo. Esto es útil cuando se quiere cambiar el estado del **calefactor** antes de que finalice un periodo de cálculo (5 s), ya que puede reaccionar antes cuando la temperatura del agua alcanza o se sale de la temperatura deseada y hacer más eficiente los cambios de estado.
   Cuando el valor del PID es cero, se apagará el calefactor, mientras que si es mayor o igual a veinte, este se mantendrá encendido durante todo el periodo de cálculo. Pero si está entre cero y veinte (ambos excluidos), el tiempo de encendido se calcula de la siguiente manera:

$$t_{subciclo} = \frac{v_{PID} \cdot t_{CONTROL}}{\text{máx}\_intervalo\_PID}$$

   Donde v_{PID} es el valor actual de controlador PID, $t_{CONTROL}$ es el periodo de cálculo del valor PID (en nuestro caso, son cinco segundos) y máx $\_intervalo\_PID$ es el límite superior del intervalo del valor PID del subciclo (20 en nuestro caso). Cuando ha pasado el tiempo del subciclo, el calefactor se apagará hasta el próximo cálculo del valor PID. El código sabe que hay un subciclo de trabajo activado cuando la variable booleana `activarSubciclo` está en `true`. Cuando el tiempo del subciclo ha pasado, esta vuelve a ser `false` hasta el siguiente periodo de cálculo del valor PID.
   En la sección 7.3.1, se harán pruebas comparativas del controlador PID cambiando sus tres constantes para observar cómo el tiempo de respuesta del calefactor es afectado cuando se acerca o se aleja de la temperatura deseada.

7.1.2. Medición del pH

El código del sensor de pH se ha extraído directamente de la página del fabricante de este, con algunas modificaciones: `https://wiki.dfrobot.com/PH_meter_SKU__SEN0161_`. El código toma muestras de la tensión analógica que emite el sensor de pH cada veinte milisegundos y hará una media aritmética de todos los valores que se han obtenido, con un desplazamiento (*offset*) determinado para calibrar el sensor de temperatura. Tras sumar ese desplazamiento y multiplicar la media aritmética de los voltajes por un factor especificado por el fabricante (3.5), se obtiene el valor del pH y se enviará al NodeMCU posteriormente con la función `enviarJSON`, que es igual al del código del sensor de la temperatura. Este código se encuentra en el fichero `SensorPH.ino`.

7.1.3. Gestión de la hora

```
DS3231 clock;
RTCDateTime dt;

void rtr_setup() {
  clock.begin();
  clock.setDateTime(__DATE__, __TIME__);
}

void rtr_loop() {
  apagar_comedero();
  dt = clock.getDateTime();
  const struct Hora horaActual = {.hora = dt.hour, .minuto = dt.minute,
      .segundo = dt.second};
  if (horaActual == horaComedero) {
    asignar_comedero(true);
  }

  digitalWrite(pin_luz, encender_luz ||
      actual_entre_dos_horas(rangoHora[0], rangoHora[1], horaActual) ?
      ON : OFF);
  enviarJSON("sensores/RTR", horaString(horaActual));
}
```

Esta parte del código del fichero `RTR.ino` tiene algunas partes destacables:

1. Se utilizan las bibliotecas `Wire.h` [27] y `DS3231.h`, que dan soporte al protocolo I2C, ya que se utilizarán dos pines especiales del Arduino: el `SDA` y `SCL`.

2. Nada más ejecutar la función `setup()` de este fichero (cada uno tiene sus propias funciones `setup()` y `loop()`), se cargará en el reloj la fecha y hora de compilación del código (macros `__DATE__` y `__TIME__`).

3. Se crea una constante del tipo `struct Hora`, que hará de soporte para gestionar la hora del sistema. Servirá para encender el comedero y encender la luz fluorescente en caso de que la hora esté dentro de un **rango de horas** especificado por la aplicación móvil. En caso contrario, esta se apagará.

4. Finalmente, la hora se enviará al servidor MQTT para actualizar la pantalla de la aplicación móvil.

7.1.4. Medición del nivel

```
#include <NewPing.h>

const unsigned int DISTANCIA_MAXIMA = 35;
bool activar_llenado = true;
NewPing HCSR04(pin_hcsr04, pin_hcsr04, DISTANCIA_MAXIMA);

void nivelAgua_setup() {
```

[27] https://www.arduino.cc/reference/en/language/functions/communication/wire/

```
8   pinMode(pin_aire, OUTPUT);
9   pinMode(pin_llenado, OUTPUT);
10  pinMode(pin_alarma, OUTPUT);
11  digitalWrite(pin_aire, OFF);
12  digitalWrite(pin_llenado, OFF);
13 }
14
15 void nivelAgua_loop()
16 {
17  apagar_comedero();
18  unsigned long nivel = HCSR04.ping_cm();
19  enviarJSON("sensores/nivel", String(nivel));
20  if (activar_llenado && nivel > nivelAlarma + 2) {
21    // Si no ha alcanzado el nivel objetivo (siempre es 2 cm menos que
        el nivel a la que se activará la alarma),
22    // la bomba se seguirá llenando.
23    digitalWrite(pin_llenado, ON);
24  } else {
25    // Si está demasiado bajo, hay que subir el nivel del agua.
26    activar_llenado = nivel >= nivelAlarma + 5;
27    seguridad_calefaccion = seguridad_bomba = !activar_llenado;
28    digitalWrite(pin_llenado, activar_llenado ? ON : OFF);
29    digitalWrite(pin_aire, activar_llenado ? OFF : ON);
30    digitalWrite(pin_bomba, seguridad_bomba && encender_bomba ? ON :
        OFF);
31  }
32
33  if (nivel <= nivelAlarma || nivel >= nivelAlarma + 6) {
34    tone(pin_alarma, 5000);
35  } else {
36    noTone(pin_alarma);
37  }
38 }
```

Este es el código completo del fichero `NivelAgua.ino`. Sus puntos a destacar son:

1. Se utiliza la biblioteca `NewPing.h` [28], que contiene una serie de métodos para medir distancias con sensores de ultrasonidos de forma **no bloqueante**. La ventaja de esta biblioteca es que al crear una instancia de la clase `NewPing`, se puede usar el mismo pin de Arduino para manejar las entradas `TRIGGER` y `ECHO` del sensor de ultrasonidos **HC-SR04**.

2. Se define una constante `DISTANCIA_MAXIMA` para establecer un límite de 35 cm de medición del sensor de ultrasonidos.

3. Si no se alcanza el nivel óptimo (2 cm por debajo del nivel de alarma), se activará la electroválvula (pin 4) hasta que llegue por primera vez a ese nivel. Cuando alcance ese nivel, se activarán el calefactor, el oxigenador de aire y la bomba de agua. En caso de que el nivel de agua esté por debajo de 5 cm del nivel de alarma, se volverá a activar la electroválvula.

4. Finalmente, si el nivel es demasiado alto (igual o superior al nivel de alarma o 6 cm

[28] `https://www.arduino.cc/reference/en/libraries/newping/`

por debajo de este), la alarma creada con el zumbador sonará hasta que se corrija el nivel anómalo.

7.1.5. Comunicación serial entre el Arduino y el NodeMCU

```
void enviarJSON(String topic, String value) {
  StaticJsonDocument<128> doc;
  doc["topic"] = topic;
  doc["value"] = value;

  String jsonString;
  serializeJson(doc, jsonString);
  miSerial.println(jsonString);
}
```

Ya se mencionó anteriormente el método `enviarJSON` en los fragmentos de código anteriores. Vamos a explicarlo:

1. La función toma dos parámetros de tipo `String`: uno para especificar el *topic* y el otro, el *payload* a enviar.

2. Se utiliza la biblioteca `NeoSWSerial.h` [29], que presenta mejoras muy importantes respecto a la que viene incluida con Arduino (`SoftwareSerial.h`), como mejor transmisión de los datos, más velocidad...
   Se utiliza la función `println`, para enviar una cadena de caracteres terminada en salto de línea para que el NodeMCU pueda procesarlos fácilmente.

3. El formato utilizado para la transmisión de los datos será `JSON`, gracias a la biblioteca `ArduinoJson.h` [30], que permite crear con facilidad objetos de tipo `JSON` para ser serializados a un objeto de tipo `String` y enviarlos cómodamente.

7.1.6. Procesamiento de datos recibidos del servidor MQTT

```
void onCharReceived(uint8_t c) {
  if (c == '\n') {
    StaticJsonDocument<128> doc;
    DeserializationError error = deserializeJson(doc, jsonString);
    if (!error) {
      String parametro = doc["parameter"].as<String>();
      if (parametro == "temperatura") {
        temperatura_deseada = doc["value"].as<double>();
      } else if (parametro == "comedero") {
        horaComedero = stringToHora(doc["value"].as<String>());
      } else if (parametro == "nivel") {
        nivelAlarma = doc["value"].as<int>();
      } else if (parametro == "rangoPH") {
        JsonArray arrayPH = doc["value"].as<JsonArray>();
        rangoPH[0] = arrayPH[0].as<double>();
        rangoPH[1] = arrayPH[1].as<double>();
```

[29] https://www.arduino.cc/reference/en/libraries/neoswserial/
[30] https://arduinojson.org/

```
      } else if (parametro == "rangoLuz") {
        JsonArray arrayLuz = doc["value"].as<JsonArray>();
        String horaInferior = arrayLuz[0].as<String>(), horaSuperior =
            arrayLuz[1].as<String>();
        rangoHora[0] = stringToHora(horaInferior);
        rangoHora[1] = stringToHora(horaSuperior);
      } else {
        String dispositivo = doc["device"].as<String>();
        bool value = doc["value"].as<bool>();
        if (dispositivo == "Comedero") {
          asignar_comedero(value);
        } else if (dispositivo == "Luz") {
          encender_luz = value;
          digitalWrite(pin_luz, value ? ON : OFF);
        } else if (dispositivo == "Bomba") {
          encender_bomba = seguridad_bomba && value;
          digitalWrite(pin_bomba, value ? ON : OFF);
        }
      }
    }
    Serial.println(jsonString);
    jsonString = "";
  } else {
    jsonString += (char) c;
  }
}
```

Esta función se invocará como si de una interrupción se tratase cuando por la comunicación serial entre el Arduino y el NodeMCU. Su funcionamiento es simple: cuando se recibe un carácter, la variable `jsonString` se irá actualizando hasta leer un **salto de línea**, donde ahí en función del parámetro se actualizará un ajuste deseado de la pecera o se activará manualmente un dispositivo, como el comedero, la bomba o la luz. Las variables donde se guardan los ajustes deseados de la pecera son:

```
// Estos serán los parámetros que serán modificados por la aplicación.
double temperatura_deseada = 30.0;                    // Temperatura
    deseada.
struct Hora horaComedero = {0, 0, 0};                 // Hora a la que
    se activará el comedero de peces diariamente.
int nivelAlarma = 5;                                  // Nivel de
    profundidad a la que se activará la alarma.
double rangoPH[2] = {0, 1};                           // Rango de pH
    deseado.
struct Hora rangoHora[2] = {{0, 0, 0}, {0, 0, 0}};    // Rango de
    encendido de la luz de la pecera.
bool seguridad_calefaccion, seguridad_bomba;          // Si el nivel
    del agua no es el óptimo, se desactivarán el calefactor y la bomba.
bool encender_luz, encender_comedero, encender_bomba; // Encendido de
    la luz, comedero y bomba de agua, respectivamente.
unsigned long tiempo_comedero;                        // Si se enciende
    el comedero, se activará este contador.
const unsigned long ENCENDIDO_COMEDERO = 7000;        // 7 s debe durar
    el comedero encendido.
String jsonString = "";                               // Búfer de
    almacenamiento de los objetos JSON recibidos.
```

El tipo `struct Hora` tiene la siguiente definición:

```
struct Hora {
  unsigned int hora, minuto, segundo;

  bool operator==(const Hora& another) const {
    return hora == another.hora && minuto == another.minuto && segundo
        == another.segundo;
  }
};
```

Todos estos fragmentos de código se encuentran en el fichero `Arduino.ino`.

7.2. Código del NodeMCU

Vamos a ver por partes el código del fichero `Pecera.ino`, que se conectará a una red Wi-Fi y luego, al bróker MQTT [13].

7.2.1. Creación de una conexión a una red Wi-Fi

```
#include <ESP8266WiFi.h>
#include <PubSubClient.h>
#include <SoftwareSerial.h>
#include <ArduinoJson.h>

SoftwareSerial mySerial(D5, D6);

// Información de la red Wi-Fi y el bróker MQTT.
const char* ssid = "Villatami_sotano"; // Nombre de la red Wi-Fi.
const char* password = "Villatami3590"; // Contraseña de la red Wi-Fi.
const char* mqtt_server = "192.168.1.51"; // IP del bróker MQTT.

// Creamos los objetos para el bróker MQTT
WiFiClient espClient;
PubSubClient client(espClient);

void establecer_conexion() {
  delay(10);
  // Nos conectamos a nuestra red Wi-Fi.
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
  }

  randomSeed(micros());
}
```

Tenemos tres variables de tipo `char*`: una para especificar el `SSID` de la red a conectarse, otra para la contraseña de la red y la otra para el nombre del bróker a conectarse. En la función `establecer_conexion()`, especificamos el modo de conexión (`WIFI_STA`) y creamos una conexión con el SSID y la contraseña anteriormente especificadas. Nótese que se utilizan las siguientes bibliotecas:

1. `ESP8266WiFi.h` [31]: Sirve para crear conexiones a una red WiFi utilizando el SDK de ESP8266 de forma sencilla utilizando el lenguaje de Arduino.

2. `PubSubClient.h` [32]: Biblioteca utilizada para crear conexiones a servidores MQTT, publicar un mensaje o suscribirse a un *topic*, crear funciones `callback` en caso de que llegue un mensaje al servidor, entre otras funcionalidades.

3. `SoftwareSerial.h`: Es la biblioteca estándar de Arduino para realizar comunicaciones seriales entre varios dispositivos usando dos pines, uno para recibir y otro para transmitir datos entre ellos. No es tan eficiente como la biblioteca `NeoSWSerial.h`, pero es la única compatible con arquitecturas distintas a la de **AVR**.

4. `ArduinoJson.h`: Sirve para transmitir datos en formato `JSON` de forma eficiente entre dispositivos.

7.2.2. La función callback

```
1  void callback(char* topic, byte* payload, unsigned int length) {
2    String payloadString = "";
3    for (unsigned int i = 0; i < length; i++) {
4      payloadString += (char) payload[i];
5    }
6    Serial.println(String(topic) + " -> " + payloadString);
7
8    StaticJsonDocument<128> doc;
9    if (String(topic) == "ajustes/tempDeseada") {
10     doc["parameter"] = "temperatura";
11     doc["value"] = payloadString.toDouble();
12   } else if (String(topic) == "ajustes/horaComedero") {
13     doc["parameter"] = "comedero";
14     doc["value"] = payloadString;
15   } else if (String(topic) == "ajustes/nivelAlarma") {
16     doc["parameter"] = "nivel";
17     doc["value"] = payloadString.toInt();
18   } else if (String(topic) == "ajustes/rangoPH") {
19     doc["parameter"] = "rangoPH";
20     JsonArray arrayRango = doc.createNestedArray("value");
21     payloadString.replace(" ", "");
22     int pos = payloadString.indexOf("-");
23     if (pos != -1) {
24       arrayRango.add(payloadString.substring(0, pos).toDouble());  //
           pH mínimo
25       arrayRango.add(payloadString.substring(pos + 1).toDouble()); //
           pH máximo
26     } else {
27       // Error en la conversión
28       arrayRango.add(-1);
29       arrayRango.add(-1);
30     }
31   } else if (String(topic) == "ajustes/rangoLuz") {
32     doc["parameter"] = "rangoLuz";
```

[31] https://esp8266-arduino-spanish.readthedocs.io/es/latest/esp8266wifi/readme.html
[32] https://www.arduino.cc/reference/en/libraries/pubsubclient/

```
33      JsonArray arrayRango = doc.createNestedArray("value");
34      payloadString.replace(" ", "");
35      int pos = payloadString.indexOf("-");
36      if (pos != -1) {
37        arrayRango.add(payloadString.substring(0, pos));  // Hora inferior
38        arrayRango.add(payloadString.substring(pos + 1)); // HoraSuperior
39      } else {
40        // Error en la conversión
41        arrayRango.add("00:00:00");
42        arrayRango.add("00:00:00");
43      }
44    } else {
45      int pos = payloadString.indexOf(" ");
46      doc["parameter"] = "control";
47      doc["device"] = payloadString.substring(0, pos);
48      doc["value"] = payloadString.substring(pos + 1).indexOf("encendid")
          != -1;
49    }
50    String jsonString;
51    serializeJson(doc, jsonString);
52    Serial.println(jsonString);
53    mySerial.println(jsonString);
54    mySerial.flush();
55    delay(250);
56  }
```

Esta función procesa el *payload* recibido de un determinado *topic* del bróker MQTT. A partir de ahí, se crea un objeto de tipo `JSON` con un determinado parámetro y valor, para ser enviado serialmente al Arduino y que este lo procese (se explicó en la subsección anterior).

7.2.3. Reconexión y bucle del cliente MQTT

```
1  void reconectar() {
2    // Bucle hasta que nos conectemos.
3    while (!client.connected()) {
4      // Creamos un identificador aleatorio.
5      String clientId = "ESP8266Client-" + String(random(0xffff), HEX);
6      // Intentamos conectarnos.
7      if (client.connect(clientId.c_str())) {
8        // Una vez conectados, lanzamos un aviso.
9        client.publish("avisos", "Conexión con el servidor MQTT
             establecida.");
10       client.subscribe("ajustes/tempDeseada");
11       client.subscribe("ajustes/horaComedero");
12       client.subscribe("ajustes/nivelAlarma");
13       client.subscribe("ajustes/rangoPH");
14       client.subscribe("ajustes/rangoLuz");
15       client.subscribe("ajustes/controlManual");
16     } else {
17       // Esperamos 5 segundos para reintentar conectarnos.
18       delay(5000);
19     }
20   }
```

```
}

void setup() {
  Serial.begin(9600);
  mySerial.begin(9600);
  establecer_conexion();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);
}

void loop() {
  if (!client.connected()) {
    reconectar();
  }

  if (mySerial.available() > 0) {
    String jsonString = mySerial.readStringUntil('\n');
    Serial.println(jsonString);
    StaticJsonDocument<128> doc;
    DeserializationError error = deserializeJson(doc, jsonString);
    if (!error) {
      client.publish(doc["topic"], doc["value"]);
    } else {
      String mensaje = "Ha habido un error al interceptar el JSON: ";
      mensaje += String(error.c_str());
      client.publish("avisos", mensaje.c_str());
    }
  }
  client.loop();
}
```

Tenemos las siguientes funciones:

1. `reconectar()`: En caso de que se pierda la conexión al bróker MQTT o sea la primera en hacerse, el NodeMCU tratará de reconectarse con un *UUID* en bucle. Tras haber logrado la conexión, enviará un mensaje de conexión exitosa al *topic* `avisos`, y luego se suscribirá a otros seis *topics*, que serán los que usará la **aplicación móvil** para transferir datos.

2. `setup()`: Creará la conexión a la red Wi-Fi e inicializará el Serial del NodeMCU y el de USB, en caso de que se quiera hacer depuraciones. También se especificará el puerto al que se conectará (por defecto, el protocolo MQTT usa el 1833 para conexiones no cifradas) y se establecerá la función *callback*.

3. `loop()`: Mantendrá el cliente conectado, a la espera de recibir o enviar datos. En el cuerpo del bucle, se espera recibir datos de los sensores de forma continua, que son enviados por el Arduino al NodeMCU por la función `enviarJSON` mencionada anteriormente.

7.3. Pruebas del sistema

Vamos a hacer algunas pruebas de ejemplo para comprobar el correcto funcionamiento del sistema.

7.3.1. Controlador PID

Se pueden cambiar las constantes del controlador PID para conseguir una respuesta más rápida a la hora de cambiar el estado de calefactor. Se han hecho cinco pruebas para determinar qué trío de constantes es el más adecuado a la hora de calibrarlo (es decir, aquel trío que haga reaccionar más rápido al calefactor a medida que se acerca o se aleja de la temperatura deseada).

Para realizar estas pruebas, es necesario desactivar las partes del código dedicadas a la hora, pH y nivel del agua, ya que solo que quiere observar el comportamiento del controlador PID ante los cambios de la temperatura del agua y poner al final del bucle principal un `delay` con 100 milisegundos, ya que cada iteración del bucle principal del código (`loop()`) es una unidad en el eje horizontal y no interesa que la gráfica vaya tan rápido, y también, porque los ejes no se pueden escalar manualmente y solo cambian su escala en función del valor de las variables graficadas.

También, es fundamental poner en `true` la variable booleana `seguridad_calefaccion`, ya que sino, el calefactor siempre estará apagado (esta variable la controla la parte del código del cálculo del nivel de agua, que activará el calefactor solo si está en un nivel adecuado).

Como hacer pruebas con el calefactor en una pecera de tamaño medio puede ser tardío, debido al volumen de agua que contiene y la temperatura varía muy poco por esta razón, es muy recomendable utilizar un acuario más pequeño o incluso una olla con el calefactor dentro y echar agua fría o caliente para realizar mejores pruebas.

Leyenda:

- **Línea roja**: Estado del calefactor (1 para encendido y 0 para apagado).
- **Línea azul**: Temperatura actual del agua de la pecera.
- **Línea verde**: Temperatura deseada del agua de la pecera.
- **Eje vertical**: Valor de las variables anteriores.
- **Eje horizontal**: Número de iteraciones del bucle `loop()`.

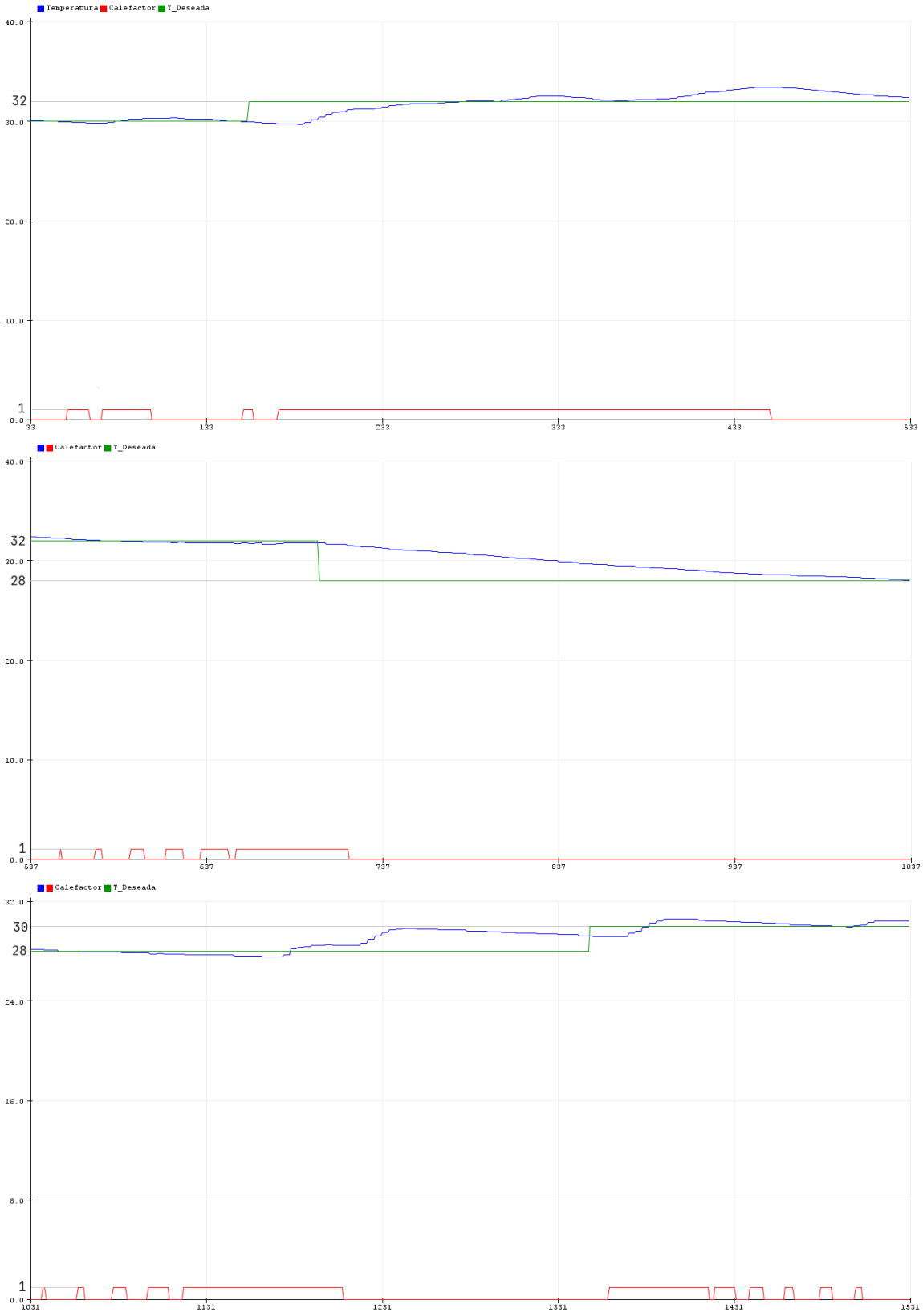

Figura 7.1: Primera prueba. Constantes: `Kp` = 15.23, `Ki` = 4, `Kd` = 1.67

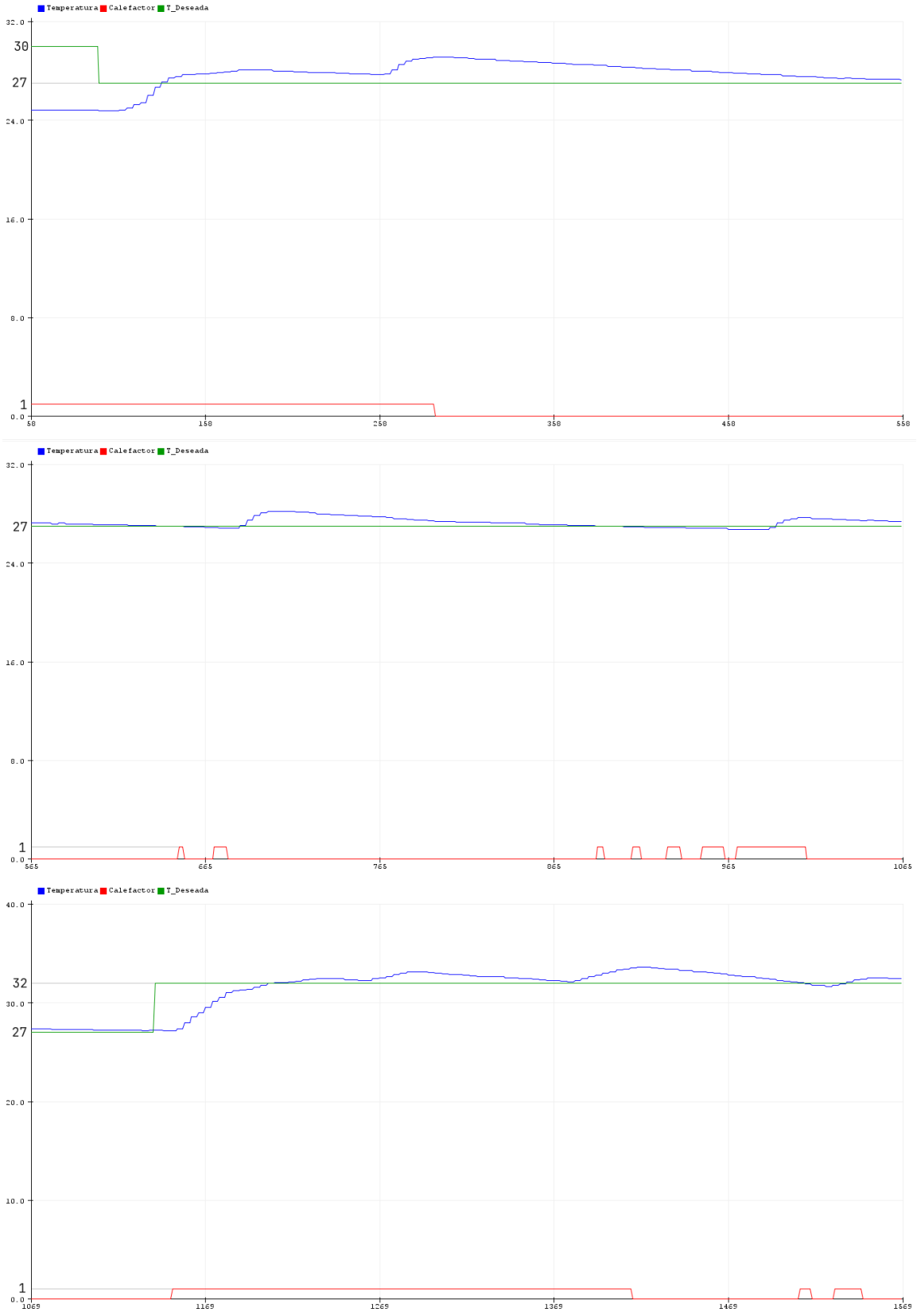

Figura 7.2: Segunda prueba. Constantes: `Kp` = 17.56, `Ki` = 4.1, `Kd` = 3

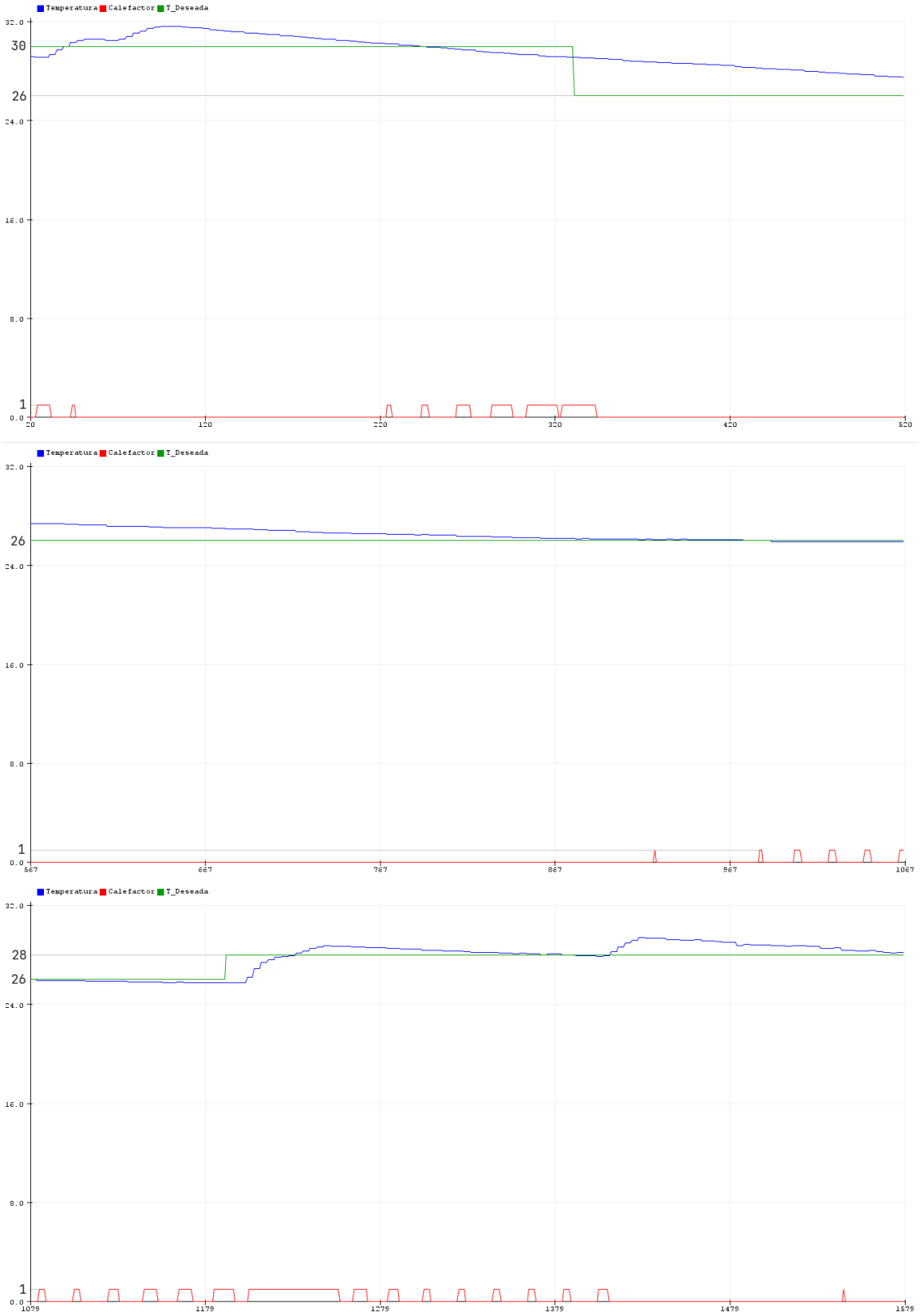

Figura 7.3: Tercera prueba. Constantes: `Kp` = 8.6, `Ki` = 1.3, `Kd` = 0.88

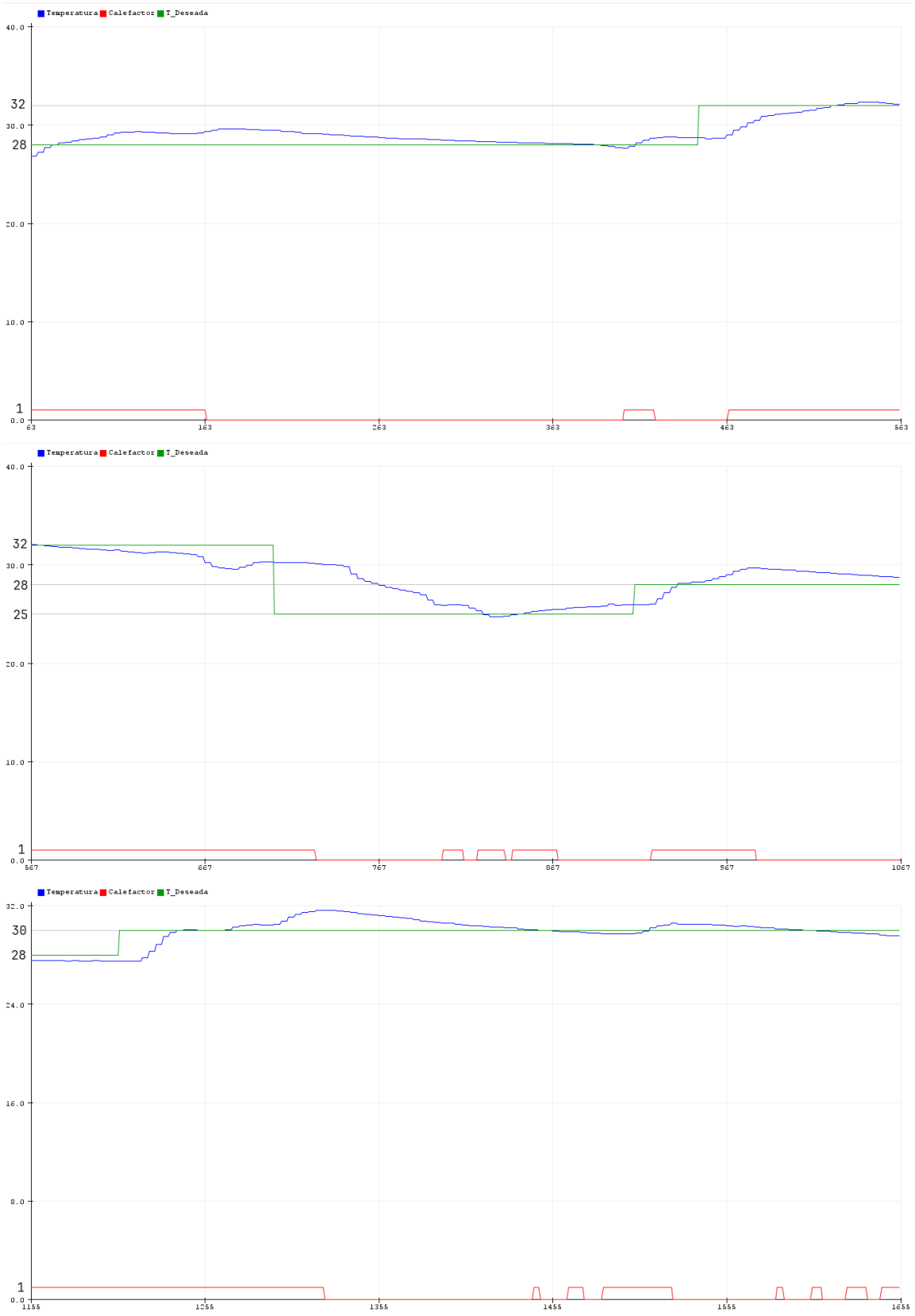

Figura 7.4: Cuarta prueba. Constantes: `Kp` = 20.34, `Ki` = 4.3, `Kd` = 9.54

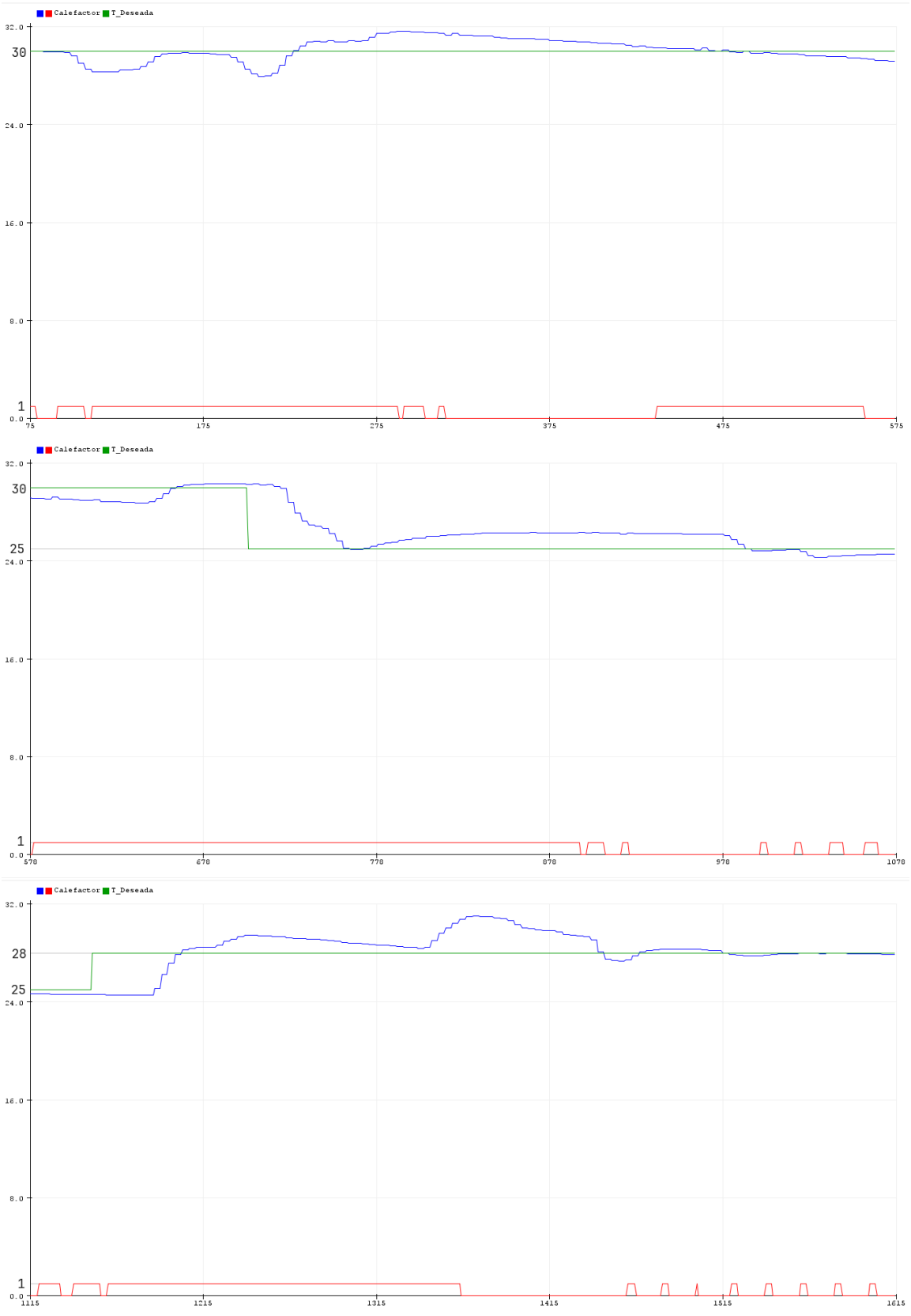

Figura 7.5: Quinta prueba. Constantes: `Kp` = 1, `Ki` = 1, `Kd` = 1

Estas gráficas han sido posibles gracias a la herramienta *Serial Plotter* que ofrece el *IDE* de Arduino [14].
El resultado de estas pruebas determina que en la **tercera prueba** el calefactor responde mejor, mientras que en la **quinta**, el calefactor responde de forma más tardía. Hacer estas pruebas es fundamental para ajustar el controlador PID, ya que cada dispositivo medidor podría requerir de constantes muy específicas. Existen varias maneras de ajustar un controlador PID: ya sea mediante ensayo y error o de una forma más precisa, con el **método Zieger-Nichols** [15].

7.3.2. Datos recibidos por el Arduino

Vamos a ver un poco del tráfico de datos que el Arduino recibe por la comunicación serial del NodeMCU (datos enviados por la aplicación móvil a este):

```
{"parameter":"temperatura","value":22.06}
{"parameter":"comedero","value":"10:15:54"}
{"parameter":"rangoLuz","value":["00:00:00","07:00:00"]}
§§o§§§{"parameter":"rangoPH","value":[7.2,7.5]}
{"parameter":"rangoPH","value":[7.2,7.5]}
{"parameter":"nivel","value":6}
{"parameter":"temperatura","value":3§}
{"parameter":"comedero","value":"00:00:00"}
{"parameter":"rangoLuz","value":["00:00:00","00:00:00"]}
{"parameter":"nivel","value":5}
{"parameter":"rangoPH","value":[0,1]}
{"parameter":"temperatura","value":20.96}
{"parameter":"rangoLuz","value":["00:00:00","06:00:00"]}
{"parameter":"control","device":"Luz","value":true}
{"parameter":"control","device":"Luz","value":false}
{"parameter":"control","device":"Bomba","value":true}
{"parameter":"control","device":"Bomba","value":false}
{"parameter":"control","device":"Comedero","value":true}
{"parameter":"control","device":"Comedero","value":false}
{"parameter":"control","device":"Comedero","value":false}
```

Figura 7.6: Datos recibidos por la comunicación serial de Arduino, visualizados en el monitor serie del puerto USB.

Aquí se pueden ciertos datos recibidos en formato `JSON`. Se pueden destacar algunos ejemplos:

- `{"parameter": "temperatura", "value": 22.06}`: Esto indica que el valor de la temperatura deseada se modificará a 22.06 ºC.
- `{"parameter": "comedero", "value": "10:15:54"}`: Esto cambiará la hora encendido diario del comedero a las 10:15:14 (se convertirá de `String` a `struct Hora` para actualizar la configuración de Arduino)
- `{"parameter": "rangoLuz", "value": ["00:00:00", "07:00:00"]}`: Cambiará el rango de horas de encendido diario de la luz fluorescente. Su nuevo rango irá desde las 0:00 hasta las 7:00.
- `{"parameter": "rangoPH", "value": [7.2, 7.5]}`: Cambiará el rango del pH aceptable del agua de la pecera a 7.2-7.5.

- `{"parameter": "nivel", "value": 6}`: Cambiará el nivel de alarma de la pecera a 6 cm.
- `{"parameter": "control", "device": "[dispositivo]", "value": [bool]}`: Activará o desactivará un dispositivo con el control manual de la aplicación en función de su valor booleano.

Nótese que a veces puede haber **pequeños errores en la comunicación**, debido a que puede no transferirse bien algún byte en el proceso de formación del `JSON`.

7.3.3. Demostración completa del sistema

El alumno ha realizado un vídeo en el que se puede ver la práctica totalidad del funcionamiento del proyecto. Se puede ver en **YouTube** con el siguiente enlace: `https://www.youtube.com/watch?v=zyszoeqqkYw`
A lo largo del vídeo, se muestran todos los componentes electrónicos en funcionamiento, luego establece algunas configuraciones deseadas para la pecera con la aplicación móvil, muestra el uso del controlador PID instalado y el minicliente de Python en acción, mostrando todos los datos que circulan por el bróker MQTT, y guardando algunas medidas de la pecera con el botón, para editarlas posteriormente.

8. Conclusiones y líneas de trabajo futuras

Con este trabajo que se ha hecho, se han aprendido nuevos conceptos, como el del **protocolo MQTT**, saber cómo hacer una aplicación móvil de forma sencilla y rápida con **Xamarin Forms** y conocer todas las posibilidades que una simple placa de desarrollo hardware, como Arduino nos puede ofrecer.
Ha habido algunas complicaciones a la hora de realizar el desarrollo del proyecto, como saber qué componentes electrónicos comprar, la curva de aprendizaje de Xamarin Forms, sobre todo por la convención **Modelo-Vista-*ViewModel*** (no confundir con el patrón **Modelo-Vista-Controlador**; tienen algunas diferencias significativas [16]), ya que es un patrón diferente a todo lo que se ha aprendido durante el grado, y también, ha habido varias dificultades a la hora de configurar el bróker MQTT, ya que constantemente daba problemas y era necesario modificar ficheros internos de la Raspberry Pi 4B para solucionar tanto la conexión SSH al terminal como la conexión a Mosquitto.
Pero con todo esto aprendido, se ha abierto una puerta a un sinfín de posibilidades, como desarrollar aplicaciones mucho más sofisticadas, utilizando, por ejemplo, un Arduino Rev R4 Wifi para simplificar circuitería y evitar usar módulos con conexión *IoT*, como el que se ha utilizado en este proyecto.
En el futuro, se podría hacer un sistema para controlar una pecera de mucho mayor tamaño y con más funcionalidades, como un **dosificador pH**, para regular automáticamente el pH del agua, sin necesidad de atención humana controlado por un servidor MQTT con más características, como que se requiera iniciar sesión y con protocolos de seguridad, como el SSH.
Hasta se podría diseñar una interfaz gráfica mediante **HTML** en el servidor para agregar funcionalidades al sistema construido, como botones para cerrar sesión y conectarse a otra cuenta con distintas credenciales, e incluso, guardar los parámetros que ha guardado un usuario en una base de datos en línea, como **mongoDB** [33].

[33] `https://www.mongodb.com/es`

Bibliografía

[1] Guellcom Acceso. Tipos de Arduino. 2022. URL: https://rayte.com/blog/post/5-tipos-de-arduino.

[2] Foro Arduino. Acuario Automatizado. 2019. URL: https://forum.arduino.cc/t/acuario-automatizado/598956.

[3] Foro Arduino. Display Acuario Arduino. 2019. URL: https://forum.arduino.cc/t/simplificar-el-codigo-con-una-funcion-pero-no-lo-logro/595382.

[4] Luis Llamas. Salidas analógicas PWM en Arduino. 2015. URL: https://www.luisllamas.es/salidas-analogicas-pwm-en-arduino/.

[5] Ángel H. ¿Cómo conectar NodeMCU con Arduino? Conexión serie UART, I2C y SPI. 2017. URL: https://borrowbits.com/2017/11/como-comunicar-arduino-con-nodemcu-parte-i-conexion-serieuart/.

[6] Carlos Barrios. Cómo instalar la Raspberry Pi 4B desde cero. 2022. URL: https://es.linkedin.com/pulse/c%C3%B3mo-instalar-la-raspberry-pi-4-desde-cero-2022-carlos-barrios.

[7] Luis Llamas. ¿Qué es MQTT? Su importancia como protocolo IoT. 2019. URL: https://www.luisllamas.es/que-es-mqtt-su-importancia-como-protocolo-iot/.

[8] Eduardo Rosas. Cómo funciona Xamarin.Forms. 2017. URL: https://platzi.com/blog/xamarin-forms/.

[9] Julio Roche. Scrum: roles y responsabilidades. URL: https://www2.deloitte.com/es/es/pages/technology/articles/roles-y-responsabilidades-scrum.html.

[10] DiagramasUML. Diagrama de casos de uso. URL: https://diagramasuml.com/casos-de-uso/.

[11] Do Your Own Robot. Diseño de circuitos electrónicos de Fritzing. 2017. URL: https://dyor.webs.upv.es/diseno-fritzing/.

[12] Naylamp Mechatronics. Tutorial RTC DS1307 y EEPROM AT24C. 2017. URL: https://naylampmechatronics.com/blog/52_tutorial-rtc-ds1307-y-eeprom-at24c.html.

[13] Luis Llamas. Cómo usar MQTT en el ESP8266/ESP32. 2021. URL: https://www.luisllamas.es/como-usar-mqtt-en-el-esp8266-esp32/.

[14] Natalia Lukianova. Cómo usar el Serial Plotter en Arduino IDE. 2022. URL: https://arduino.cl/como-usar-el-serial-plotter-en-arduino-ide.

[15] Carlos Pardo Martín. Método de Ziegler-Nichols. 2018. URL: https://www.picuino.com/es/control-ziegler-nichols.html.

[16] Adictos al Trabajo. MVC y MVVM. 2012. URL: https://www.adictosaltrabajo.com/2012/10/07/zk-mvc-mvvm/.

Apéndice A. Minicliente Python

Vamos a crear un minicliente MQTT hecho con Python. Debemos seguir los siguientes pasos (utilizando **Windows**):

1. Descargar la última versión de `Python`: `https://www.python.org/downloads/`.

2. Seguir los pasos del asistente de instalación (es importante agregar `python.exe` al PATH al inicio de la instalación).

3. Instalar el paquete `paho-mqtt`. Para ello, escribir el siguiente comando en un terminal: `pip install paho-mqtt`.

4. Una vez instalado, abrimos cualquier *IDE* de Python. Para este manual, utilizaremos `IDLE`.

5. Importamos los módulos `paho-mqtt` y `uuid`: para ello, escribimos en un fichero de extensión `.py` lo siguiente:

   ```
   import paho.mqtt.client as mqtt
   import uuid
   ```

6. Después, nos debemos conectar al bróker MQTT en línea con un identificador (se usa *UUID* v4). Para ello, escribimos las siguientes líneas de código:

   ```
   client_uuid = str(uuid.uuid4())
   cliente = mqtt.Client(str(uuid.uuid4()))
   cliente.connect("IP o nombre del bróker")
   ```

7. Nos suscribimos a los *topics* deseados. Para ello, usamos el método `subscribe`:

   ```
   cliente.subscribe(topic="topic1")
   cliente.subscribe(topic="topic2")
   cliente.subscribe(topic="topic3")
               .
               .
               .
   ```

8. Creamos funciones **callback** para hacer determinadas tareas cuando el cliente logre conectarse al servidor o cuando este recibe mensaje de un *topic* al que se ha suscrito el cliente:

   ```
   def on_connect(client: mqtt.Client, userdata, flags, rc):
       if rc == 0:
           client.publish(topic="avisos",
   ```

```
                        payload=f"Cliente Python conectado con éxito.
                         (UUID: {client_uuid})")
    else:
        print(f"Error al conectarse al servidor MQTT.
                (Código de error: {rc})")


def on_message(client, userdata, message):
    print(f"{message.topic} -> {message.payload.decode("utf-8")}")


cliente.on_connect = on_connect
cliente.on_message = on_message
```

También podríamos publicar mensajes a un determinado *topic* con el método `publish`, como se hace en el fragmento de código de arriba.

9. Creamos el bucle para mantener la conexión activa:

```
try:
    cliente.loop_start()

    while True:
        pass
except KeyboardInterrupt:
    cliente.disconnect()

cliente.loop_stop()
```

Lo que se acaba de realizar es un minicliente funcional de Python para leer todos los mensajes que se reciben de un bróker MQTT. Se podría agregarle más funcionalidades, pero para este proyecto, es más que suficiente con esto.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA