Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de un sistema de monitorización conectado para la medición del nivel de CO_2 en espacios cerrados

Autor: Raúl Alejandro Ochoa Aguilar

Tutor: José Manuel Lanza Gutiérrez

2021

UNIVERSIDAD DE ALCALÁ
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Trabajo Fin de Grado

Desarrollo de un sistema de monitorización conectado para la medición del nivel de CO_2 en espacios cerrados

Autor: Raúl Alejandro Ochoa Aguilar

Tutor: José Manuel Lanza Gutiérrez

Tribunal:

Presidente: Nombre del presidente del tribunal

Vocal 1: Nombre del primer vocal

Vocal 2: Nombre del segundo vocal

Calificación: .....

Fecha: .....

Agradecimientos

Quiero agradecer, en primer lugar a mis padres y a mi hermana que han estado durante toda mi vida dándome apoyo y guiándome para lograr mis sueños. Sé que estarán muy orgullosos con mi graduación.

A mi novia, Irene, por creer siempre en mí y por apoyarme en todo momento, compartiendo conmigo todos los avances del proyecto aunque en muchas ocasiones no entendiese nada de lo que estaba hablando.

Me gustaría agradecer también a José Manuel Lanza Gutiérrez, mi tutor, por haberme guiado a lo largo del proyecto y por ayudarme siempre con todas mis dudas.

Gracias a toda mi familia y amigos que de una forma directa o no han ayudado a conseguir mis objetivos.

Resumen

Se ha desarrollo un sistema informático con el fin de monitorizar el nivel de $C0_2$ en espacios cerrados bajo la tecnología IoT. Este sistema consta de un sistema servidor y un sistema sensor.

La información acerca del $C0_2$ es proporcionada por una colección de sistemas sensores programados para ejecutar un cliente publicador del protocolo MQTT, mediante el cual comunicarse con el servidor.

En el sistema servidor se guardan, en una base de datos, los datos enviados por los sensores. Estos datos pueden ser consultados mediante una interfaz web, que además permite labores de administración acerca del despliegue de los sensores y sus ubicaciones.

Palabras clave: CO_2, IoT, MQTT, *Docker*, *Kubernetes*, *MongoDB*.

Abstract

A computer system has been developed in order to monitor the level of $C0_2$ in enclosed spaces using IoT technology. This system consists of a server system and a sensor system.

Information about $C0_2$ is provided by a collection of sensor systems programmed to run an MQTT protocol publishing client, through which to communicate with the server.

The data sent by the sensors is stored in a database on the server system. This data can be consulted through a web interface, which also allows administration of the deployment of the sensors and their locations.

Palabras clave: CO_2, IoT, MQTT, *Docker*, *Kubernetes*, *MongoDB*.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción al Trabajo Fin Grado

Este apartado tiene como objetivo introducir al lector en la motivación que llevó a realizar el proyecto. Además, se definen los objetivos del mismo, la solución propuesta, y por último, se detalla la estructura de la memoria.

1.1. Motivación

La monitorización de la calidad del aire en los espacios cerrados es un tema de gran relevancia para la sociedad, debido a que en la actualidad, los habitantes de las ciudades de los países industrializados pasan el 90 % de su tiempo en este tipo de espacios con ventilación limitada, ya sea para trabajar, estudiar o vivir [7]. Así, la salud de las personas dependerá, en gran medida, de la calidad del aire interior (IAQ, del término anglosajón *Indoor Air quality*) que respiran [8]. Por ello, la monitorización de esta información ha atraído una gran atención, siendo más intensa en el último año debido a la pandemia mundial provocada por el coronavirus SARS-CoV-2[9].

Como es bien sabido, la principal vía de contagio del coronavirus SARS-CoV-2 se produce en espacios interiores debido a los aerosoles exhalados por una persona contagiada, tenga o no síntomas [10]. Una medida aceptada acerca de la acumulación de aerosoles en un espacio cerrado es el nivel de CO_2. Así, cuando el nivel de CO_2 se encuentra por encima de un cierto umbral medido en partes por millón (ppm, del término anglosajón *Parts Per Million*), se recomienda encarecidamente la ventilación del área, ya sea por medios naturales o mecánicos [11].

Otro de los problemas relacionados con la calidad del aire interior es el síndrome del edificio enfermo [8]. Este síndrome, que se relaciona con un conjunto de diferentes síntomas que no suelen estar acompañados de ninguna enfermedad orgánica o signos físicos, se manifiesta principalmente en las personas que se encuentran en estas edificaciones. Hasta el momento, no ha sido posible averiguar la causa, o al menos, determinar el origen exacto de estos síntomas, pero lo que si es seguro, es de que tiene una clara relación con la mala calidad del aire interior[8].

Una tecnología de reciente desarrollo y aplicación, como el Internet de las Cosas (IoT, del término anglosajón *Internet of Things*)[12], permite realizar despliegues de sensores heterogéneos conectados de relativo bajo coste. Por ello, esta tecnología podría utilizarse para monitorizar la calidad del aire en estos espacios interiores, especialmente el nivel

de CO_2, combinando sensores de distinta tipología. Así, este proyecto se centra en el desarrollo de un sistema de monitorización del CO_2 de bajo coste apoyado en tecnología IoT.

1.2. Objetivos

El objetivo principal de este proyecto consiste en desarrollar un sistema de monitorización conectado para la medición del nivel de CO_2 en espacios cerrados no industrializados, que permitirá conocer en tiempo real la concentración de este gas. Para la consecución de dicho objetivo, se requiere alcanzar diversos subobjetivos:

- Definir la arquitectura del sistema, compuesta por una serie de sistemas sensor y un único sistema servidor (con su correspondiente *front-end* y *back-end*). Los sistemas sensor se encargarán de medir periódicamente la concentración de CO_2, la cual será enviada al sistema servidor. El sistema servidor será el encargado de recolectar la información procedente de los sensores para ser mostrada al usuario. Además, permitirá gestionar el sistema de monitorización.

- Implementar las tecnologías por desplegar en el sistema servidor. Como por ejemplo, bases de datos, receptores de mensajes en protocolos específicos de IoT e interfaces de usuario.

- Implementar las tecnologías por desplegar en el sistema sensor. Como por ejemplo, conectividad inalámbrica y protocolos específicos de IoT para el envío de información.

El campo de aplicación del proyecto se enfoca hacía el uso del sistema de monitorización en espacios interiores de reducidas dimensiones (por ejemplo, viviendas, aulas de clase u oficinas), donde sea interesante medir el nivel de CO_2. No sería adecuado su uso en centros de investigación (laboratorios) o entornos industriales, ya que los sensores utilizados, al ser de bajo coste, no tendrían el nivel de especificación requerido. Por otro lado, tampoco sería adecuado para su uso en espacios cerrados de grandes dimensiones, por las propias limitaciones de especificación de estos sensores de bajo coste. Sin embargo, el proyecto sería susceptible de ampliar su campo de aplicación a los nombrados previamente si se elimina la restricción de coste, o si bien, la tecnología de sensorización avanza, pudiendo asumir mediciones de mayor precisión con coste limitado.

1.3. Solución propuesta

La solución propuesta en este proyecto consiste es un sistema de monitorización conectado para la medición del nivel de CO_2 en espacios cerrados. Para conseguir esto, se ha utilizado una placa sensor conectada físicamente a una placa de desarrollo (un nodo de IoT). Ambas placas conforman un componente que denominaremos elemento sensor, sistema sensor o sensor, que será el encargado de enviar los datos capturados por la placa sensor al servidor mediante la placa de desarrollo. Para ello, se hará uso de un protocolo de comunicación habitual en el IoT como MQTT (del término anglosajón *Message Queuing Telemetry Transport*).

En el sistema servidor, los datos enviados por el elemento sensor son procesados por el componente denominado *Broker* MQTT, que actúa como una oficina de correos, recibiendo todos los datos que están siendo publicados por el cliente (el elemento sensor), permitiendo que la información esté disponible para los subscriptores (un componente del servidor llamado controlador). El controlador ejerce como cliente subscriptor del protocolo de comunicación MQTT, procesando los datos dejados por el sensor en el *Broker* MQTT para luego guardarlos en la base datos con una estructura defina.

Los datos capturados por los sensores en relación con el CO_2 presente en cada espacio podrán ser visualizados mediante una interfaz web, la cual proporciona advertencias en el caso de superar un nivel máximo establecido y permite acceder a los datos históricos registrados. Los usuarios administradores además también podrán administrar todos los sensores desplegados y los espacios donde se encuentren desplegados, pudiendo añadir y eliminar tanto los espacios como los sensores a demanda.

1.4. Estructura de la memoria

Este documento cuenta con distintos capítulos en los que se recoge la información más relevante acerca del desarrollo llevado a cabo en este proyecto, así como los fundamentos teóricos requeridos para la comprensión de esta memoria. Cada uno de estos apartados se describe brevemente a continuación:

1. **Introducción:** En este primer capítulo se introduce al lector en la motivación para la realización del proyecto, se definen los objetivos y se detalla la solución propuesta, así como la estructura de esta memoria.
2. **Base teórica y selección de tecnologías:** Se exponen los fundamentos teóricos requeridos para una correcta compresión de esta memoria. Además, se lleva a cabo una selección y descripción de las tecnologías utilizadas en el proyecto.
3. **Diseño del sistema:** Se realiza una descripción de los distintos componentes que componen el sistema a nivel de diseño, así como de la estructura del modelo de datos.
4. **Implementación del sistema:** Se describe los detalles más relevantes acerca de la implementación del sistema servidor y el sistema sensor.
5. **Conclusiones:** Se exponen las principales conclusiones obtenidas durante el desarrollo del sistema.
6. **Líneas futuras:** Se analizan las líneas futuras de trabajo, incluyendo posibles mejoras del sistema.
7. **Bibliografía:** Se lista la bibliografía utilizada para el desarrollo de este proyecto, la cual es convenientemente citada a lo largo del documento.
8. **Anexo I:** Se expone el presupuesto para el desarrollo del proyecto.
9. **Anexo II:** Se detalla el manual de usuario del sistema, incluyendo los detalles requeridos para el despliegue del sistema servidor y la programación del sistema servidor. Además, se proporciona un manual de uso para el entorno web.

Capítulo 2

Base teórica y selección de tecnologías

Este apartado tiene como objetivo exponer los conceptos principales acerca de las tecnologías utilizadas en el sistema, así como su base teórica, además de realizar un análisis del estado del arte en relación a soluciones para la monitorización del CO_2. Concretamente, se comenzará con la revisión del estado del arte, para después abordar los temas de él dióxido de carbono y la calidad del aire, el IoT, las bases de datos y la contenerización.

2.1. Estado del arte

Existe una gran variedad de productos comerciales utilizados para medir la concentración de CO_2 en espacios interiores, algunos de los cuales solo proporcionan los datos medidos en el propio dispositivo ya sea por una pantalla o indicadores LED, otros además cuentan con plataformas para visualizar los datos de los sensores. Muchos de estos productos solo cuentan con la posibilidad de desplegar un único sensor, ya que no cuentan con una forma de comunicar más de un sensor con el servidor. En cambio, otros cuentan con la posibilidad de desplegar más de un sensor, pudiendo hacer mediciones de CO_2 en edificios enteros. A continuación, en las secciones siguientes, se describirán las características de dos productos comerciales para la medición del CO_2. Como se ha introducido previamente, el número de productos disponibles es amplio, por lo que estos dos productos servirán como referencia de lo ofrecido en el mercado. Finalmente, se proporciona una comparativa de estos productos con el proyecto a desarrollar.

2.1.1. Medidor de CO_2 DIOXCARE PDF

DIOXCARE PDF[1] es un medidor de CO_2 que detecta el dióxido de carbono en el medio ambiente en tiempo real. Véase la Figura 2.1. Cuenta con un sensor NDIR (véase la Sección 2.3.3), basado en el principio de absorción de la luz infrarroja para detectar el CO_2. El precio del dispositivo es de 119,00 € y cuenta con las siguientes características:

- El sensor mide el CO_2, la temperatura y la humedad. Además, cuenta con una pantalla LCD de 3,2 pulgadas y con una alarma, que en caso de exceder los niveles configurados, se activa. También se puede configurar el registro de datos de medición por intervalo de tiempo.

[1] https://protect.soiartdistribucion.com/producto/medidor-de-co%E2%82%82-dioxcare/

Figura 2.1: Medidor de CO_2 *DIOXCARE* PDF.[1]

- Cuenta con una aplicación para ordenador que permite almacenar y visualizar toda la información de las mediciones vía USB, además permite exportar los registros en PDF, ver gráficos de tendencias o histórico.

El sensor cuenta con las siguientes características técnicas:

- Tipo de sensor: NDIR para medición de CO_2
- Rango de medición en concentración de CO_2: 0 9999 ppm.
- Rango de medición en temperatura: -20 60 ℃.
- Rango de medición en humedad: 0-100 % RH.
- Fuente de alimentación: Batería de 3,7V recargable (ion de litio de 2200mAh) o alimentada por un cable USB conectado a una fuente de 5V.

2.1.2. Medidor de CO_2 CO2Panel

CO2Panel[2] es un sistema de medición de CO_2 que permite medir este gas en las ubicaciones que se necesiten. Éste permite la instalación por el propio usuario, ya que está calibrado y listo para su puesta en marcha. CO2Panel dispone de dos sensores, uno es un sensor llamado CO2Panel PI (véase la Figura 2.2), que mide los niveles de CO_2 en el aire. Éste dispone de un indicador LED para conocer mediante el color cuando los niveles del gas superan los definidos por CO2Panel [2]. Otro sensor es el CO2Panel *MATRIX*, que cuenta con un sensor y un panel LED diseñado para mostrar el nivel de CO_2. Ambos cuentan con un sensor NDIR (véase la sección 2.3.3).

Estos dos dispositivos, aparte de mostrar las variaciones del CO_2 mediante un indicador LED o panel LED, permiten consultar los datos mediante el móvil, ordenador o

[2] https://co2panel.shop/

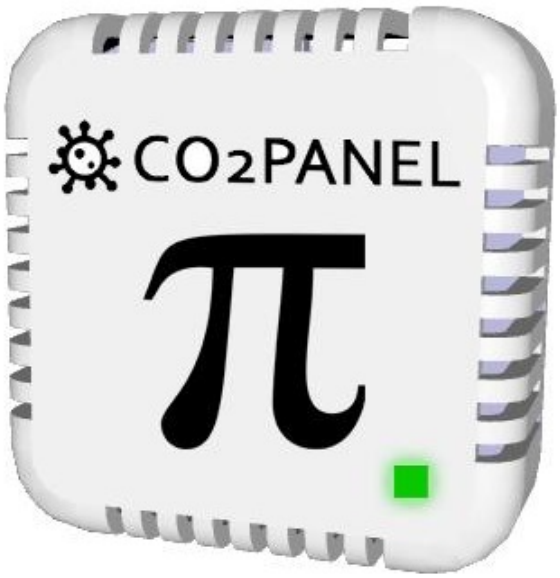

Figura 2.2: Medidor de CO_2 *CO2Panel* PI.[2]

smart TV, ya que cuentan con una conexión a internet por WiFi. La conexión a internet permite enviar los datos a la nube para poder consultarlos a través del CO2Panel *cloud*, a la que envían las mediciones de CO_2 cada minuto.

CO2Panel *cloud* es una plataforma que permite gestionar los sensores mencionados mediante una aplicación web, accediendo a ésta con un usuario y una contraseña. Esta plataforma permite acceder al histórico y consultar los registros de CO_2 a lo largo de 24h. Además permite visualizar los datos de un modo gráfico, pudiendo descárgalos en formatos EXCEL o JSON. Las configuraciones que se pueden llevar a cabo en la plataforma son las de nombre, email y contraseña para el acceso a ésta, la configuración de alertas por SMS o e-mail según un umbral de CO_2 y la calibración de los sensores en remoto. La plataforma cuenta con dos modalidades, básico y prémium. La modalidad básica es gratuita, permitiendo acceder a los datos de los sensores, pero no puediendo ver todos al mismo tiempo ni compartir los datos a través de una *smart TV*. La modalidad prémium, con un coste de 1€/dispositivo al mes, permite ver todos los sensores en un mismo panel y compartir la información a través de una *smart TV*.

CO2Panel *cloud* permite la integración con otras plataformas como IFTTT (del término anglosajón *If This, Then That*) para la integración de los sensores con otros accionamientos o servicios, como sistemas domóticos, sistemas de ventilación o de control de apertura y cierre de ventanas cuando los niveles de CO_2 aumentan.

El precio del sensor CO2Panel PI es de 129,00€ y cuenta con las siguientes características:

- Tipo de sensor: NDIR para la medición de CO2
- Rango de medición de concentración de CO_2: 0-5000ppm.
- Alimentación: 5V - 2A (adaptador de corriente incluido).

- Tiempo de medición(s): Cada 5 segundos, enviando los datos cada minuto al servidor.
- Interfaz Gráfica: Led RGB (verde, amarillo, rojo) con 7 niveles de intensidad.
- Conexión a internet: Vía WiFi
- Humedad de operación: 0 % - 85 %

2.1.3. Comparativa del estado del arte con este proyecto

Entre los dos productos comerciales descritos anteriormente y el proyecto existen varias diferencias como se puede ver en el Cuadro 2.1 y que se describirán a continuación. En primer lugar, las diferencias entre el sensor *DIOXCARE* PDF y el proyecto son las siguientes:

- Una de las principales diferencias entre ambos productos, que no se refleja en la tabla, es que el en el sistema *DIOXCARE* PDF no pueden desplegarse múltiples sensores, ya que el sensor no cuenta con conexión a internet y solo se puede extraer la información de éste vía USB. En cambio en el proyecto, que cuenta con conexión a internet vía WiFi, se podrá desplegar más de un sensor logrando cubrir los espacios que se necesiten monitorizar bajo una única solución.
- El tipo de sensor es otra diferencia importante, porque el sensor *DIOXCARE* PDF utiliza NDIR y el sensor del proyecto utiliza MOS, por lo que el sensor *DIOXCARE* PDF obtiene mediciones más precisas y con rangos mayores que van desde 0 a 9999ppm, mientras que el sensor del proyecto obtiene rangos que van desde 0 a 1000ppm. Esta diferencia de rangos no toma tanta importancia en espacios interiores porque, como se verá en la Sección 2.2, los niveles normales de CO_2 en estos espacios es menor a 500ppm. Además, el diseño realizado en este proyecto es fácilmente ampliable a sensores de mayor precisión.
- Otra diferencia es que el sensor *DIOXCARE* PDF cuenta con sensor de temperatura mientras que el utilizado en el proyecto no tiene. Igualmente, resultaría fácilmente ampliable el sistema para incluir la monitorización de temperatura.
- La principal diferencia entre estos dos sistemas, más allá del despliegue multisensor, está en el precio, siendo el precio del dispositivo *DIOXCARE* PDF igual a 119€ y siendo de 11,35€ el coste base (precio del *hardware*, sin ganancia) del sistema sensor utilizado en este proyecto (véase la Sección A.1), por lo que la diferencia es amplia.

Las principales diferencias entre el sensor *CO2Panel PI* y el considerado en este proyecto son las siguientes:

- Los dos cuentan con una plataforma que permite gestionar los sensores. La principal diferencia es que *CO2Panel PI* cuenta con una plataforma desplegada en un *cloud* propietario de la marca. Mientras que la plataforma del proyecto puede ser desplegada en cualquier servidor o servicio de *cloud* que permita la contenerización. Además, todas las funcionalidades de la plataforma del proyecto son gratuitas no siendo así en la plataforma *CO2Panel PI* como se vio antes. Cabe destacar que la plataforma *CO2Panel PI* permite realizar configuraciones en remoto de los sensores no siendo así en el proyecto.

- La diferencia en los sensores utilizados está en que *CO2Panel PI* utiliza sensores de tipo NDIR y el sensor del proyecto utiliza MOS. Como se describió previamente, utilizar NDIR como tipo de sensor permite obtiene mediciones más precisas y con rangos mayores, que en este caso van de 0 a 5000ppm, un rango inferior al proporcionado por *DIOXCARE* PDF. Otra diferencia es que el sensor *CO2Panel PI* cuenta con sensor de temperatura mientras que el utilizado en el proyecto no tiene. Ambos sensores cuentan con conexión a internet vía WiFi y permiten despliegues multisensor.

- Por último, nuevamente la diferencia más relevante entre ambos sistemas se encuentra en el coste, siendo mucho mayor el del sensor *CO2Panel PI*, ascendiendo a los 129€, en comparación con el precio base del sistema sensor de 11,35€. Esta diferencia de coste permitiría desplegar un mayor número de sensores a un menor coste, aunque a una precisión limitada, como ya se introdujo previamente.

Características	**DIOXCARE PDF**	**CO2Panel PI**	**Proyecto**
Precio por unidad	119€	129€	11,35€
Tipo de sensor	NDIR	NDIR	MOS
Rango de medición del CO_2	0 - 9999 ppm	0 - 5000ppm	10 - 1000ppm
Rango de medición temperatura	-20°C 60 °C	0°C 50°C	-
Conexión a internet	-	WIFI	WIFI

Cuadro 2.1: Comparativa entre dos productos comerciales y el sistema desarrollado en este proyecto.

Este análisis de mercado permite cuantificar la aportación de este proyecto a los dispositivos de medición de CO_2. Así, se ha comprobado que el proyecto aporta una solución económica y competitiva a los productos existentes, aportando características o superiores a productos con mayor coste. Además la solución propuesta en este proyecto permite desplegar la plataforma en un servicio de *fog* o de *cloud*, pudiendo cambiar según las necesidades.

2.2. El dióxido de carbono y la calidad del aire

El dióxido de carbono, cuya fórmula química es CO_2 [13], es un gas sin olor e incoloro compuesto por carbono y oxígeno. Es el cuarto gas más abundante en la atmósfera terrestre con una concentración aproximada de 400 ppm [14]. Además, es el principal gas que provoca el efecto invernadero, afectando directamente al problema del calentamiento global [15]. Este gas se origina de forma natural en diferentes procesos, principalmente por la respiración de los organismos vivos, aunque también es generado por la contaminación humana, como por ejemplo debido al transporte motorizado, la industria o la ganadería intensiva [15].

El dióxido de carbono es uno de los contaminantes que más afectan a la salud humana. Si una persona se expone a concentraciones cercanas a los 30.000 ppm, puede afectar al cerebro causando dolor de cabeza, falta de concentración, mareos y problemas respiratorios. En el caso de superar los 30.000 ppm puede provocar la asfixia por desplazamiento

Nivel	**Concentración máxima de CO_2, en ppm**
IDA 1	350
IDA 2	500
IDA 3	800
IDA 4	1.200

Cuadro 2.2: Clasificación acerca de la calidad del aire en cuatro niveles, según la concentración de $C0_2$. [6]

del oxígeno [16].

Acorde a la normativa para la calidad del ambiente interior en edificios de uso público de la Comunidad de Madrid (establecida en base al Real Decreto 1027/2007, de 20 de julio y modificado por Real decreto 238/2013). Los niveles de CO_2 de 300 a 400 ppm en ambientes exteriores y de 600 a más de 2000 ppm en ambientes internos no son considerados tóxicos ni contaminantes, pero se consideran indicadores de la calidad del aire debido a que la principal fuente de emisiones en interiores son las propias personas. En los espacios interiores, cuando los niveles de CO_2 se encuentran entre los 800 y 1.200 ppm las personas pueden comenzar a sentir incomodidad, cansancio, dolores de cabeza y problemas respiratorios. En estos espacios cerrados, los efectos graves se producen a partir de los 5.000 ppm pudiendo llegar a producir desvanecimientos.

Acorde a esta normativa de la Comunidad de Madrid, el Departamento de Salud Ambiental de la Subdirección General de Salud Pública del Ayuntamiento de Madrid emitió un informe técnico sobre la correcta ventilación de los edificios como medida preventiva a la infección por SARS-Cov-2 debido a aerosoles [6]. En este informe, se concreta que la calidad del aire se podría cuantificar en cuatro niveles habitualmente relacionados con ciertas localizaciones. Esta clasificación se resume en el Cuadro 2.2, donde se indica el valor máximo de concentración para cada uno de los niveles y se detalla a continuación:

- IDA 1 (aire de óptima calidad): hospitales, clínicas, laboratorios y guarderías.
- IDA 2 (aire de buena calidad): oficinas, residencias (locales comunes de hoteles y similares, residencias de ancianos y de estudiantes), salas de lectura, museos, salas de tribunales, aulas de enseñanza y asimilables y piscinas.
- IDA 3 (aire de calidad media): edificios comerciales, cines, teatros, salones de actos, habitaciones de hoteles y similares, restaurantes, cafeterías, bares, salas de fiestas, gimnasios, locales para el deporte (salvo piscinas) y salas de ordenadores.
- IDA 4 (aire de calidad baja).

Este informe técnico indica además que la calidad del aire debe mantenerse en niveles IDA 1 e IDA 2 para tener una calidad buena, es decir, igual o por debajo a 500ppm. Además, concentraciones mayores de 500 ppm y menores que 800ppm se identifican como de categoría de calidad media. Por último, concentraciones superiores a 800ppm se consideran como de calidad mala. Esta subclasificación en relación a la calidad del aire buena, media y mala se resume en el Cuadro 2.3.

Calidad del aire	Concentración de CO_2, en ppm
Buena	<= 500
Media	500 < ppm <= 800
Mala	> 800

Cuadro 2.3: Clasificación acerca de la calidad del aire en tres niveles, según la concentración de $C0_2$.

Con la aparición del coronavirus SARS-Cov-2, la monitorización de la calidad del aire en espacios interiores, acorde a los niveles de CO_2, toma gran relevancia, tomando medidas a nivel de gubernamental para evitar la propagación de este virus en espacios cerrados o de poca ventilación. Esto se indica en el documento publicado por el Ministerio de Industria, Comercio y Turismo y el Ministerio de Sanidad acerca de las *recomendaciones de operación y mantenimiento de los sistemas de climatización y ventilación de edificios y locales para la prevención de la propagación del SARS-Cov-2*. [17]. En este documento, se indica que el riesgo de transmisión aérea del virus Sars-Cov-2 en los edificios es mayor cuando la ventilación es insuficiente. Para evitar esto se recomienda el uso de sistemas especializados en la medición del CO_2, como el desarrollado en este proyecto. Además, el documento indica tener como referencia la clasificación IDA2 como medida de buena calidad del aire. Así, en caso de superar tal nivel se recomienda aumentar la ventilación o reducir la ocupación del espacio interior que lo supere.

En consecuencia de esta regulación, se ha considerado que el sistema desarrollado en este proyecto debe generar una alerta visual cuando el nivel de concentración de $C0_2$ se encuentre en los ya comentados niveles de media y mala calidad del aire. Esto es, cuando la concentración sea superior a 500ppm.

2.3. El Internet de las Cosas (IoT)

El IoT puede definirse como el proceso que permite conectar objetos físicos a Internet, integrando software, sensores y otras tecnologías con el fin de contar con una red de elementos que interactúan entre sí [12]. Véase la Figura 2.3. Así, el IoT permite que multitud de dispositivos de reciente tecnología (como televisores, bombillas, enchufes, altavoces, frigoríficos o aspiradores) puedan interactuar entre ellos y con las personas que tienen en su entorno, dotándolos en algunos casos de una cierta inteligencia, la cual nos facilita el día a día. Es tal el impacto del IoT, que se cree que para finales del año actual (2021) habrá cerca de 25 billones de dispositivos interconectados[18]. El ecosistema IoT, dado la gran cantidad de dispositivos conectados de distinta topología, no se tiene que entender como un ente homogéneo, sino como un conjunto diverso de conceptos, protocolos y tecnologías, enfocados en un mismo objetivo: la interconectividad y el procesamiento distribuido de la información[19].

Habitualmente, el IoT suele dividirse en torno a tres capas: el *cloud* o la nube, el *fog* o la niebla y el *edge* o el extremo. En el *edge*, se encuentran todos aquellos dispositivos cercanos al usuario que se encargan de capturar información en su entorno y/o interactuar con éste. Los dispositivos en esta capa suelen estar constreñidos en coste (para permitir

Figura 2.3: El ecosistema del internet de las cosas.

el despliegue de un gran número de dispositivos), capacidad energética (uso de baterías para facilitar y economizar su despliegue) y capacidad de cómputo (para reducir el coste y permitir compatibilidad con el uso óptimo de baterías). En el *fog*, se encuentran dispositivos que permiten realizar procesados más ambiciosos sobre los datos capturados en el *edge*, estando habitualmente cercanos a estos últimos dispositivos del *edge*, lo que permite reducir el tiempo de respuesta. Algunos ejemplos de dispositivos en el *fog* podrían ser enrutadores, teléfonos inteligentes y pequeños servidores. Finalmente, en el *cloud*, se encuentran los grandes centros de cómputo, con una enorme capacidad de cálculo, pero situados habitualmente a gran distancia de donde la información es capturada, lo que implica mayores tiempos de reacción del sistema y posibles problemas de confidencialidad y seguridad de la información.

En este proyecto, se propone el uso de dispositivos desplegados en el *edge* (el sensor o sensores que permitirán cuantificar el nivel de CO_2 en el ambiente), así como un sistema servidor que permitirá recolectar la información y administrar el sistema. Este sistema servidor podría localizarse en el *fog* o el *cloud*, atendiendo a los requisitos del despliegue realizado.

Las siguientes subsecciones detallan el protocolo de comunicación estándar en IoT, llamado MQTT, y la plataforma IoT de código abierto denominada *NodeMCU*.

2.3.1. El protocolo de comunicación MQTT

El protocolo de comunicación MQTT (del término anglosajón *Message Queing Telemetry Transport*)[20] es un conocido protocolo máquina a máquina (M2M, del término anglosajón *Machine to Machine*)[21] para mensajería estándar en IoT, llegando a ser uno

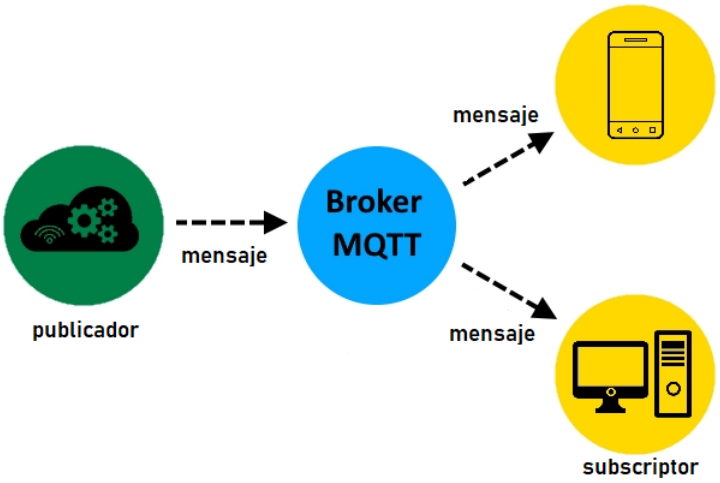

Figura 2.4: Esquema de funcionamiento del protocolo MQTT.

de los más populares es este entorno [20].

El éxito de este protocolo se basa en su sencillez, ligereza y un uso del ancho de banda mínimo. Estas características unidas a que la gran mayoría de los dispositivos del *edge* del IoT están constreñidos en potencia, consumo, y/o ancho de banda, lo convierten es uno de los más utilizados. En la actualidad, se utiliza en una amplia variedad de sectores, como la automotriz, la fabricación, las telecomunicaciones, el petróleo y el gas [20]

La base de la comunicación es el protocolo TCP/IP [22]. MQTT funciona como un servicio de mensajería donde un publicador emitirá mensajes de una temática o *topic* determinado en un punto intermedio denominado *broker* MQTT. Por otro lado, los subscriptores a un *topic* determinado se conectarán al *broker* para acceder a los mensajes publicados en esa temática. Véase la Figura 2.4. Este esquema publicador-subscriptor tiene inherentes las ventajas de escalabilidad, asincronismo y desacomplamiento [20].

Las siguientes subsecciones detallan, en primer lugar, algunas consideraciones relacionadas con MQTT y la seguridad. A continuación, se introduce la herramienta *software* seleccionada para hacer uso del protocolo MQTT en este proyecto.

2.3.1.1. MQTT y la seguridad

La seguridad es unos de los aspectos más importantes y críticos de un protocolo de comunicación. De forma nativa, MQTT no cuenta con consideraciones de seguridad, enviando la información sin cifrar. Sin embargo, es posible implementar medidas de seguridad en otras capas de la comunicación, como transporte SSL/TLS [23], autenticación por usuario y contraseña o mediante certificado.

Para la utilización de una de estas medidas de seguridad, hay que tener en cuenta que los dispositivos IoT, en su mayoría, tienen una baja potencia y memoria, por lo que la utilización de una de estas medidas de seguridad basadas en *software* pueden generar una sobrecarga o mal rendimiento de los dispositivos. Sin embargo, existen también soluciones

en el mercado que incluyen chips criptográficos[3] para los nodos del *edge* a cambio de un mayor coste, realizado el proceso de encriptado sin afectar a la capacidad de cómputo principal del dispositivo.

2.3.1.2. Mosquitto: una implementación abierta de MQTT

Eclipse Mosquitto [24] es un servidor o mediador de mensajes de código abierto (con licencia EPL/EDL) que implementa el protocolo MQTT. Este servidor es uno de los más utilizados en el mundo IoT. En el protocolo de comunicación MQTT, se colocaría como el agente o *broker* MQTT encargado de recibir los mensajes enviados por los publicadores y distribuirlos entre los subscriptores ellos. La página oficial se puede encontrar aquí [4].

Mosquitto es muy ligero, adecuado tanto para dispositivos de baja potencia como servidores completos. Además, es multiplataforma por lo que es altamente portátil y con una rápida implementación en una gran variedad de dispositivos. Cuenta con una gran comunidad, lo que le aporta un gran soporte. También cuenta con bibliotecas como *Paho* que proporciona implementaciones en una amplia variedad de lenguajes de programación [24].

En lo relativo a la seguridad, *Mosquitto*, en su última actualización (Versión 2.0) exige a los usuarios que decidan activamente cómo configurar la seguridad en su *broker*. Así, se requiere que los usuarios definan explícitamente en el documento de configuración la seguridad del *broker*, en caso contrario éste solo funcionara de forma local en el dispositivo que lo ejecute. Las opciones de seguridad que se pueden implementar en el *broker* mediante el fichero de configuración son las siguientes:

- Configurar el acceso al *broker* mediante autenticación con usuario y contraseña.
- Configurar la comunicación con el *broker* mediante seguridad de la capa de transporte (TLS, del término anglosajón *Transport Layer Security*).

2.3.2. NodeMCU: una plataforma IoT de código abierto

El término NodeMCU [25], se refiere una plataforma de desarrollo de propósito general orientada al *edge* del IoT. Esta plataforma se basa en una placa de desarrollo de bajo coste que consta de un microcontrolador *ESP8266* y un *firmware Open Source*, el cual utiliza el lenguaje de programación *Lua*.

Para comprender las posibilidades de la plataforma *NodeMCU*, primero deben conocerse los detalles técnicos del microcontrolador *ESP8266*[5]. Este chip consta de las siguientes características:

- Alimentación a 3.3VDC.
- Procesador *Tensilica Xtensa* LX106 80 MHz (160 MHz *overclock*) con un núcleo de procesamiento.

[3] https://www.microchip.com/DevelopmentTools/ProductDetails/DM320118
[4] https://mosquitto.org/
[5] https://docs.ai-thinker.com/_media/esp8266/docs/esp-12f_product_specification_en.pdf

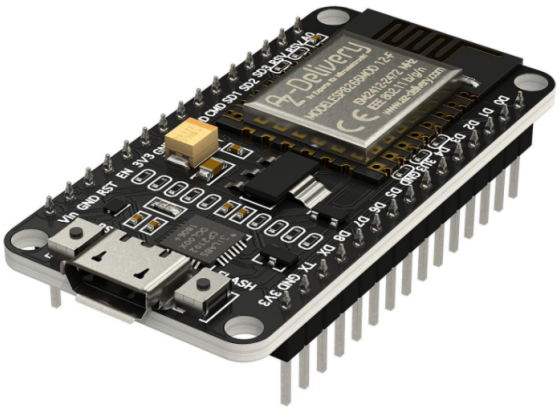

Figura 2.5: Placa de desarrollo *AZ-Delivery NodeMCU Lolin*[3].

- Frecuencia de reloj de CPU de 80 MHz.
- Memoria RAM de 64 KB de instrucción y 96 KB de datos.
- Entradas mediante 16 pines GPIO (Entradas/Salidas de propósito general).

En este proyecto, se va a hacer uso de la placa de desarrollo *AZ-Delivery NodeMCU Lolin* en su versión V3. Véase la Figura 2.5. Esta placa consta de las siguientes características:

- Utiliza la interfaz de comunicación en serie USB CP2102.
- Utiliza una versión actualizada del microcontrolador *ESP8266*, con denominación *ESP8266MOD-12F*, lo cual le otorga adicionalmente de conectividad WiFi 2.4GHz 802.11b/g/n y aumenta ligeramente la capacidad de la memoria ROM con 128KB y RAM de 4MB. Este microcontrolador consta de la distribución de pines mostrada en la Figura 2.6. De los 30 pines, 17 pines son GPIO(3,3V 15mA max), un pin es entrada analógica para el ADC, 5 pines son GND, 3 pines son alimentaciones (de salida) con 3,3v, un pin VUSB con salida de 5v, un pin Vin que permite alimentar el microcontrolador con 5v y un pin RST para reiniciar el microcontrolador.
- Soporta seguridad WPA y WPA2 y tiene el protocolo TCP/IP integrado.
- Programación de la placa a través de un cable micro USB-B.
- Alimentación con 5V mediante micro USB-B.
- Cuenta con el chip CH340 para la interfaz USB.

La placa de desarrollo puede programarse con código *Lua* a través del puerto USB incluido, usando para ello el IDE de *Arduino*[6] o el *software PlatformIO*[7]. Esta última herramienta es la utilizada en este proyecto. *PlatformIO* es una herramienta multiplataforma que permite trabajar sobre una gran cantidad de microcontroladores, plataformas y *frameworks* diferentes, de una forma intuitiva, sencilla e integrada. Esta herramienta permite además abstraerse de las configuraciones que permiten inicializar el microcontrolador, lo que puede considerarse como una interesante ventaja si no se requiere un gran

[6] https://www.arduino.cc/en/software
[7] https://platformio.org/

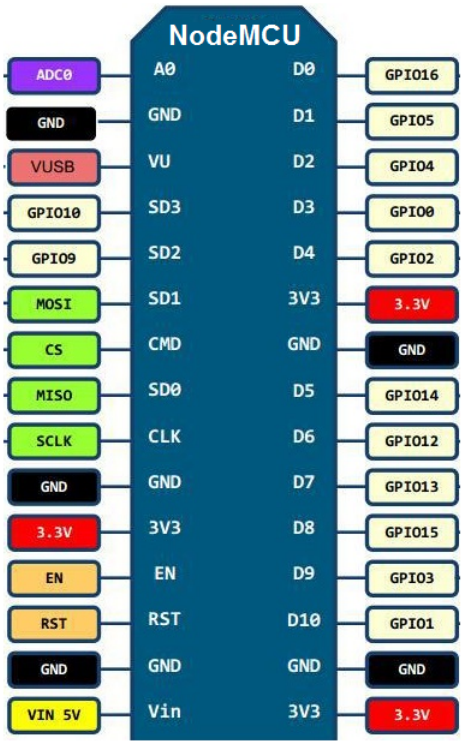

Figura 2.6: Distribución de pines del microcontrolador *ESP8266MOD-12F* [3].

nivel de personalización en ese aspecto. También permite depurar, compilar y visualizar varios dispositivos en paralelo. *Platformio* cuenta con un gestor de dependencias, que permite administrar bibliotecas de *PlatformIO Registry*[8] y repositorios VCS (Git, Hg, SVN). El gestor facilita la búsqueda, la instalación y el mantenimiento de bibliotecas actualizadas.

Por otro lado, se encuentran disponibles distintas bibliotecas listas para ser usadas con este microcontrolador. Las más importantes utilizadas en este proyecto son:

- *ESP8266WiFi*[9]: Permite conecta el *NodeMCU* a una red Wifi. Esta biblioteca ha sido desarrollada basándose en el SDK del ESP8266 y en WiFi.h de la biblioteca *Arduino* WiFi *Shield*.

- PubSubClient v2.8 [10]: Permite ejecutar el *NodeMCU* como un cliente MQTT, pudiendo enviar y recibir mensajes MQTT. Es compatible con versiones MQTT a partir de la v3.1.

2.3.3. Selección del detector de CO_2 para IoT

Los detectores de CO_2 son instrumentos relativamente asequibles usados para medir la concentración de este gas. Habitualmente, este tipo de instrumento no solo identifica y cuantifica la presencia de CO_2, sino también de otros compuestos o partículas, lo que puede dar lugar a limitaciones en los sistemas desarrollados. Las principales tecnologías usadas habitualmente a tal efecto son:

[8] https://platformio.org/lib
[9] https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266WiFi
[10] https://platformio.org/lib/show/89/PubSubClient

- Infrarrojos no dispersivos (NDIR, del término anglosajón *Non Dispersive Infrared Detector*)[26]: Tecnología que utiliza longitudes de onda de luz específicas para medir la concentración de CO_2. La ventaja de esta tecnología es que la presencia de otras sustancias no altera la lectura. Sin embargo, ésta puede verse afectada por la humedad o la temperatura ambiente.

- Sensores electroquímicos: Utilizan la corriente eléctrica o la conductividad para medir la concentración de CO_2. La ventaja de ta tecnología es la menor susceptibilidad a los cambios de humedad y temperatura. La desventaja es que otra sustancias en el aire pueden alterar la lectura.

- Sensores semiconductores de óxido metálico (MOS, del término anglosajón *metal-oxide-semiconductor*): Utilizan la resistividad de compuestos metálicos para medir la concentración de CO_2. La ventaja de esta tecnología es que tiene un diseño muy simple, lo que hace que sea fácil de usar. La desventaja es que la lectura puede verse afectada por la temperatura y la humedad, así como por la presencia de otras sustancias.

En este proyecto el sensor seleccionado para la detección de CO_2 es el MQ-135 (hoja de catálogo[11]). Este sensor, de la serie MQ[27], utiliza tecnología MOS para medir la concentración de CO_2. La principal ventaja que aporta ese sensor es su bajo coste (véase la Sección A.1, donde se detalla el presupuesto del proyecto), así como su compatibilidad a la hora de poder ser utilizado junto a placas de desarrollo como la seleccionada en este proyecto. Su principal limitación es que es sensible a la presencia de otros compuestos. Sin embargo, tal y como se detallará más abajo, los compuestos identificables por este sensor (más allá del CO_2), no son habituales en el tipo de entorno objetivo de este proyecto.

Los sensores de la serie MQ poseen una capa sensible constituida por un óxido metálico, material resistivo cuyo valor de resistencia es sensible al gas que lo recubre. La diferencia en la resistividad del material indica si está presente un gas en particular y en que concentración. Este tipo de sensor cuenta con un elemento calefactor que permite que el sensor se encuentre a una temperatura estable, de modo que el valor resistivo obtenido en relación a la concentración del gas sea independiente de la temperatura. Esto implica que en necesario esperar un tiempo determinado por el fabricante a que tal elemento calefactor adquiera la temperatura adecuada, una vez se ha proporcionado alimentación al sensor. Este tiempo depende del modelo y suele estar entre 12 y 48 horas.

Los sensores de la serie MQ generalmente se proporcionan en módulos o placas de circuito impreso, simplificando la conexión y facilitando su uso. Así, tan solo se requiere alimentar el módulo, esperar el tiempo de estabilización definido por el calefactor y comenzar a recolectar la información proporcionada. En este proyecto, se ha considerado la placa sensor que contiene el MQ-135, que se observa en la Figura 2.7. La placa sensor cuenta con las siguientes especificaciones:

- Voltaje de operación de 5V DC.

- Corriente de operación de 150mA.

[11] https://www.olimex.com/Products/Components/Sensors/Gas/SNS-MQ135/resources/SNS-MQ135.pdf

Figura 2.7: Sensor MQ135 montado sobre una placa de circuito impreso[4].

- Potencia de consumo de 800mW.
- Resistencia de carga ajustable mediante potenciómetro.
- Tiempo de precalentamiento o estabilización mayor a 24h.
- Rango de detección de partes por millón de 10ppm 1000ppm.
- Concentración detectable de los gases amoniaco, dióxido de nitrógeno, alcohol, benceno, dióxido y monóxilo de carbono.

El placa sensor cuenta con 4 pines como se puede ver el la Figura 2.8 donde:

- *Analog out*: Es un pin analógico cuyo valor es un nivel de tensión en el rango 0-5v, que se encuentra relacionado con la concentración del gas. Este pin será conectado en el proyecto a la entrada ADC del *NodeMCU*, para realizar una conversión de este valor de tensión a un nivel discreto en el rango 0 a 1023 (acorde a la resolución del ADC presente en el *NodeMCU*). Esta conversión será necesaria ya que los sistemas digitales (el microcontrolador y su núcleo de procesamiento) requieren que la información por utilizar sea también de tipo digital.
- *Digital out*: Es un pin digital de tipo TTL[28], por lo que tiene únicamente dos estados, *LOW*, con un rango de tensión comprendido entre 0,0V y 0,8V, y *HIGH*, con un rango de tensión comprendido entre 2,2V y 5v. Este pin se pone a nivel *LOW* cuando la concentración detectada se encuentra por encima de un valor establecido mediante el potenciómetro, y *HIGH* en caso contrario. En este proyecto, se ha optado por utilizar la aproximación *software* (uso del ADC), puesto que no requiere de modificar el *setup hardware* (potenciómetro) para modificar el comportamiento del sistema.
- *Vcc(+5v)*: Es el pin de alimentación del sensor a 5v.
- *Ground*: Es el pin de referencia de tensión (tierra).

Acorde a la hoja de catálogo del sensor, éste presenta la sensibilidad para detectar los distintos gases según la Figura 2.9. En este gráfico, se puede ver la concentración de los gases en ppm según la relación de resistencia del sensor (Rs / Ro). Donde, Rs es la resistencia del sensor que cambia según la concentración del gas y Ro es la resistencia del sensor a una concentración conocida en ausencia de otros gases.

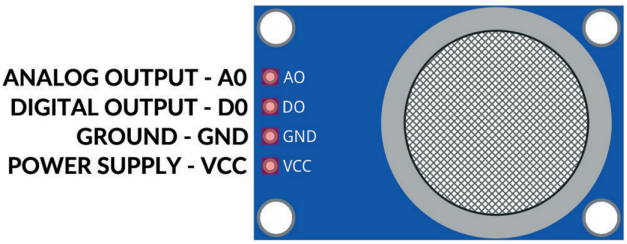

Figura 2.8: Distribución de pines de la placa con el sensor MQ-135[4].

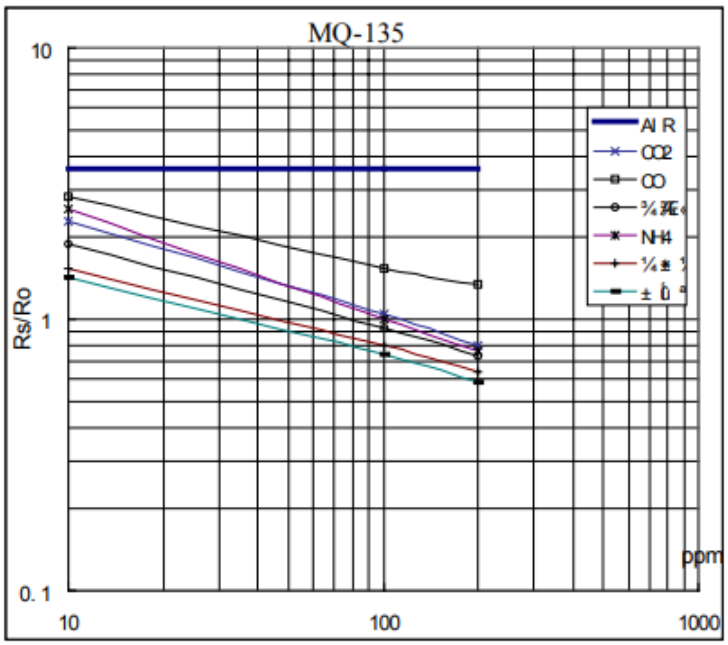

Figura 2.9: Sensibilidad del MQ-135 para la detección de gases. Figura obtenida directamente de la hoja de catálogo[5].

2.4. Bases de datos

Los datos y la correcta manipulación de estos son un punto importante en cualquier servicio tecnológico. En los últimos años, el 90 % de los líderes de negocios hablan de los datos como uno de los recursos clave para cualquier negocio [29].

Las bases de datos son una serie de datos o información estructurada que pertenecen a un mismo contexto, que son almacenados de forma organizada para su posterior uso [30]. Estos datos son usualmente tratados con un sistema de gestión de bases de datos (DBMS, del término anglosajón *Database Management System*) que permite el almacenamiento de forma ordenada y una recuperación rápida y eficaz de los datos.

Las bases de datos pueden clasificarse de distintas formas, por ejemplo:

- Bases de datos relacionales: Se organizan en forma de tablas con filas y columnas. Esta tipología permite acceder a información estructurada de forma eficiente y flexible.

- Bases de datos orientadas a objetos: La información se representa en forma de

Base datos relacional	Base datos no relacional
Tablas	Colecciones
Filas	Documentos

Cuadro 2.4: Equivalencia entre una base de datos relacional y una no relacional.

objetos.

- Bases de datos distribuidas: Se organizan de forma distribuida, con una serie de archivos situados en diferentes ubicaciones.

- Bases de datos NoSQL: Las bases de datos NoSQL o no relacionales permiten que los datos no estructurados o semi estructurados se almacenen. A diferencia de las bases de datos relacionales, no se define que estructura deben tener los datos por almacenar.

- Bases de datos orientadas a grafos: Almacenan datos en términos de entidades y las relaciones entre éstas.

En este proyecto se ha optado por utilizar bases de datos no relacionales con el propósito de almacenar documentos compuestos con la información proporcionada por los sensores. Las ventajas que aportan las bases de datos no relacionales al proyecto, en comparación con las relacionales, es que éstas soportan un mayor volumen de datos. Además, no requieren de identificadores, lo que permite establecer relaciones entre diferentes conjuntos de datos y ofrecer mayor flexibilidad y escalabilidad horizontal. Estas ventajas encajan con el propósito del proyecto, ya que la flexibilidad y escalabilidad son una de las características principales de todo desarrollo en IoT.

Como se puede ver en el Cuadro 2.4, la equivalencia entre una base de datos relacional y una no relacional consiste en que las relacionales se basan en tablas, que equivalen a colecciones en una no relacional. Además, en las relacionales las tablas están compuestas por filas (las distintas entradas de datos), que equivalen a los documentos en las no relacionales. Concretamente en el proyecto, se ha optado por utilizar la conocida base de datos *MongoDB*, que se describe en la siguiente subsección.

2.4.1. MongoDB

MongoDB es una base de datos NoSQL, distribuida, de código abierto, multiplataforma y basada en documentos [31].

En *MongoDB*, los datos se almacenan en formato JSON (acrónimo del término anglosajón *JavaScript Object Notation*) [32]. Los documentos JSON admiten campos incrustados, por lo que los datos relacionados y las listas de datos se pueden almacenar en un mismo documento en lugar de en una tabla externa. Un ejemplo de uso del formato JSON en pares de (nombre / valor) se encuentra en la Figura 2.10.

Las principales características de *MongoDB* son la velocidad, como la más importante, el poder realizar todo tipo de consultas, la indexación, que a diferencia de la mayoría de las bases de datos relacionales, cualquier campo puede ser indexado. Además de que

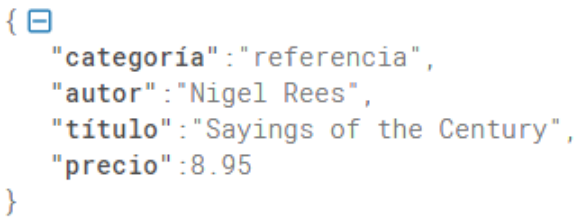

Figura 2.10: Ejemplo de formato JSON.

cuenta con balanceo de carga (es la capacidad de ejecutarse en múltiples servidores, balanceando la carga o replicando los datos para poder mantener el sistema funcionando en caso que exista un fallo), almacenamiento de archivos y la capacidad de realizar consultas utilizando *JavaScript* [31].

Otra característica importante es que *MongoDB* permite crear un conjunto de servidores que mantienen la misma copia de datos, llamado conjunto de réplicas o *replica set*. El funcionamiento de los *replica set* consiste en que los nodos donde se guardan los datos, se clasifican en nodo primario o *primary node* y nodos secundarios o *secondary nodes*. Los nodos secundarios se encargan de mantener las copias actualizadas mediante un mecanismo asíncrono, logrando de esta forma, mantener la misma información en todos los nodos. Esta separación en nodos permite que en caso de que un nodo primario falle, uno de los nodos secundarios pasaría a ser el primario, asegurando el acceso a los datos. Esta característica es fundamentar para un proyecto de IoT como éste, ya que asegura el acceso a los datos en caso de caídas.

También cuenta con otras ventajas como la validación de documentos, motores de almacenamiento integrado y menor tiempo de recuperación ante fallos. Las desventajas son que no es una solución para aplicaciones con transacciones complejas, además de que no tiene un reemplazo para las soluciones de herencia.

2.5. Contenerización

La contenerización es un procedimiento de virtualización a nivel de sistema operativo que permite implementar y ejecutar aplicaciones distribuidas sin tener que lanzar una máquina virtual completa para cada aplicación. Este procedimiento permite tener varios sistemas aislados ejecutándose en un único *host* que accede a un único *kernel*. Los contenedores, que agrupan procesos ejecutándolos de forma nativa, contienen todos los componentes necesarios para ejecutar el *software* deseado. Estos componentes pueden ser archivos, variables de entorno o bibliotecas.

La ventaja de la contenerización con respecto a la virtualización es que mejora la eficiencia de la memoria, la CPU y el almacenamiento. Dado que estos no tienen la sobrecarga de una máquina virtual, se pueden ejecutar más contenedores utilizando la misma infraestructura. Otro beneficio es la portabilidad, siempre que la configuración entre sistemas sea la misma, el contenedor puede ejecutarse en cualquier sistema o nube sin cambiar el código. Una desventaja de la contenerización es la falta de aislamiento del sistema operativo central, lo que provoca que los contenedores sean vulnerables si el sistema operativo

central no es seguro.

La contenerización replantea cómo los equipos de desarrollo de *software* pueden hacer un uso más eficiente de los recursos computacionales, llegando a ejecutar aplicaciones en contenedores en lugar de utilizar máquinas virtuales, convirtiendo la contenerización en una práctica común en la industria del *software*.

Un concepto importante en la contenerización son los orquestadores de contenedores. Estos permiten controlar y dirigir la creación de contenedores, verificar su correcta ejecución y ofrecer una correcta gestión de errores. Los principales proveedores de servicios de contenerización y orquestadores de contenedores son los siguientes:

- ***Docker Platform*** [12]**:** Conjunto de productos de plataforma como servicio (PaaS) de código abierto que permite a los usuarios empaquetar, distribuir y administrar aplicaciones dentro de contenedores. El componente principal es el *software Docker* y las herramientas principales son *Docker Compose* (herramienta utilizada para definir y ejecutar aplicaciones *Docker* de varios contenedores) y *Docker Swarm*(Orquestador de contenedores).

- ***Kubernetes Engine*** [13]**:** Sistema de código abierto para la implementación, expansión y gestión automatizada de aplicaciones en contenedores.

- ***OpenShift*** [14]**:** Es una plataforma de contenedores como servicio (PaaS) proporcionada por *Redhat*. Permite la automatización de aplicaciones en recursos seguros y escalables en la nube. Además proporciona plataformas para crear, implementar y administrar aplicaciones en contenedores.

- ***Elastic Container Service (ECS) de Amazon*** [15]**:** Plataforma de organización de contenedores que permite a los clientes de *Amazon Web Services* (AWS) ejecutar aplicaciones en contenedores, permitiendo contenedores *Docker*.

- ***Nomad*** [16]**:** Orquestador de contenedores proporcionado por *HashiCorp*. Este orquestador proporciona cargas de trabajo simples y flexibles para implementar y administrar contenedores.

En este proyecto se ha optado por utilizar la contenerización de aplicaciones con objetivo de ejecutar todos los componentes de software del sistema servidor en contenedores. Concretamente en el proyecto, se ha optado por utilizar el *software Docker*, como plataforma de contenedores, y *Kubernetes*, como orquestador de contenedores de *Docker*.

En líneas generales, las ventajas que aporta *Kubernetes* al proyecto en comparación con los otros orquestadores es que éste utiliza su propia API, con la ventaja, de que se pueden añadir nuevas operaciones personalizadas según se necesiten en el proyecto. Además, permite trabajar con distintos tipos de contenedores como *docker*, *rkt*, *cri-o* o *frakti*. Otra ventaja es la alta disponibilidad de éste mediante la supervisión de nodos,

[12] https://www.docker.com/
[13] https://cloud.google.com/kubernetes-engine?hl=es-419/
[14] https://www.openshift.com/
[15] https://www.amazonaws.cn/en/ecs/
[16] https://www.nomadproject.io/

detectando los que no funcionan correctamente y permitiendo eliminarlos. Por otra parte, *Docker* proporciona al proyecto el poder crear y ejecutar contenedores, así como almacenar y compartir imágenes de contenedor. Además de que *docker* está muy extendida, bien documentada y lo más importante, es fácil de usar. Ambas, *Kubernetes* y *Docker* se describen en las siguientes subsecciones.

2.5.1. Docker

Docker es una plataforma que permite separar las aplicaciones de su infraestructura [33]. Con *Docker*, se puede empaquetar y ejecutar cada módulo de un sistema en un entorno aislado llamado contenedor. Con esta forma de encapsular aplicaciones en contenedores, se puede tener varios contenedores ejecutándose simultáneamente gracias al aislamiento y la seguridad de estos. Dentro del ecosistema de un contenedor, se contiene todo lo necesario para ejecutar la aplicación concreta. La página oficial se puede encontrar aquí [17].

Con la utilización de *Docker*, se logra agilizar el ciclo de vida del desarrollo, dotando a los desarrolladores de un entorno estandarizado. Se pueden desplegar los desarrollos en contenedores locales, virtuales en centros de datos o con proveedores de nube. Así, al utilizar *docker*, se permite que una aplicación pueda ser escalable y modular a lo largo de su vida. Dada la naturaleza liviana de *docker*, éste permite la administración dinámica de cargas de trabajo, ampliando o eliminando aplicaciones y servicios según la necesidad.

Docker utiliza una arquitectura cliente-servidor, donde el cliente es un programa que se ejecuta en el ordenador del usuario y el servidor es un *daemon* o demonio, un servicio que gestiona los contenedores (véase la Figura 2.11). El cliente representa la línea de comandos (CLI, del término anglosajón *command-line interface*) por la cual los usuarios gestionan el ciclo de vida de las imágenes, redes, volúmenes y contenedores. El cliente recoge los comandos escritos por los usuarios, para después comunicarse con el servidor vía API *Rest* pasándole el comando para que éste lo ejecute. El servidor es el proceso principal de *Docker*, que se encarga de gestionar los objetos existentes en el servidor, los cuales pueden ser contenedores o los siguientes componentes:

- Imágenes: Son instancias de un contenedor. Éstas funcionan como una plantilla (que incluyen una aplicación y sus bibliotecas) que se utilizan para construir contenedores.
- Redes: Permiten que los contenedores se conecten mediante una red.
- Volúmenes: Espacio donde se almacenan los archivos. Éstos, al ser gestionados por *Docker*, permiten que el resto de procesos del *host* no puedan acceder a tal información.

Como se puede ver en la Figura 2.12, para construir un contenedor en *docker*, primero se tiene que crear el *dockerfile* (documento de texto que contiene todos los comandos necesarios para construir una imagen para *docker*) de la aplicación que se quiere *dockerizar*, para luego crear la imagen de la aplicación. Por último, se crea el contenedor a partir de la imagen ejecutando ésta. Para crear una imagen y un contenedor se utiliza una CLI de *docker*.

[17] https://https://www.docker.com/

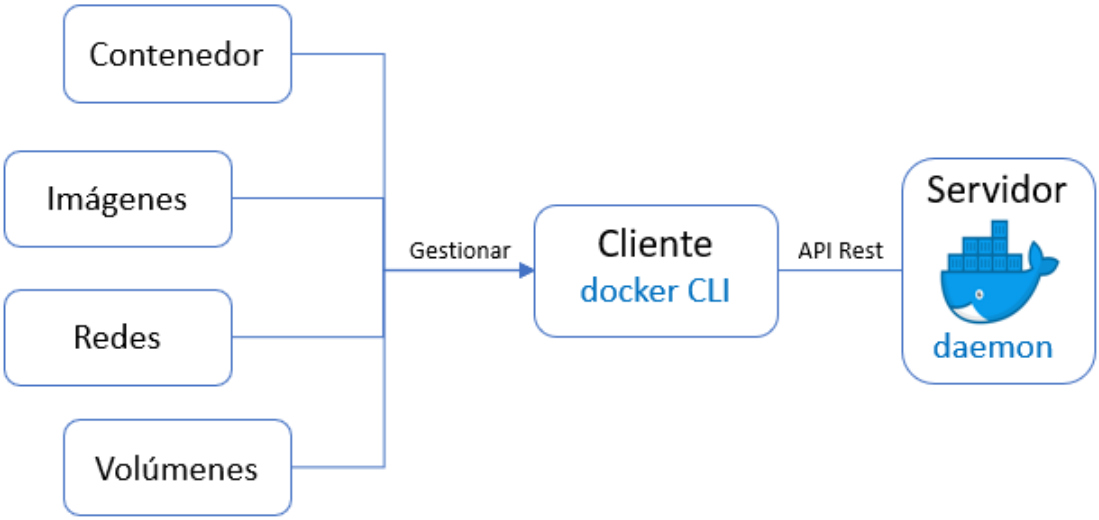

Figura 2.11: Arquitectura de *Docker*.

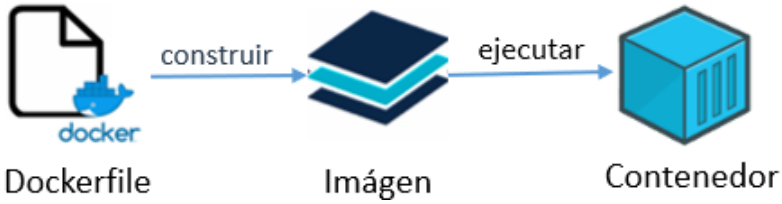

Figura 2.12: Flujo de construcción de un contenedor en *docker*.

Docker permite crear imágenes propias o utilizar imágenes ya existentes. El objetivo del proyecto es crear una imagen por cada componente del sistema. *MongoDB* y *Mosquitto* son piezas de *software* que ya cuentan con imágenes en *Docker Hub*[18](servicio utilizado para buscar y compartir imágenes de contenedores). Por lo que en el proyecto solo se crearan imágenes de los componentes restantes. Estos componentes tienen implementaciones diferentes del *dockerfile* para crear las imágenes, ya que están programados en lenguajes de programación diferentes. En el proyecto se utilizará *Java* [19](lenguaje de programación orientado a objetos y una plataforma informática) para crear el componente controlador y *React*[20](biblioteca de *JavaScript* para crear interfaces de usuario) para crear el cliente y el servidor web. Los pasos para crear una imagen por cada lenguaje se describen a continuación:

- *Java*: Para crear una imagen *java* en *docker* es necesario crear la aplicación usando el *framework Spring*[21], que permite generar un jar (archivo que permite ejecutar una aplicación de *Java*) de la aplicación y luego crear el archivo de configuración *dockerFile*. En el archivo *dockerFile*, se indica la versión del JDK (*software* que proporciona herramientas de desarrollo para la creación de programas en *Java*) mediante el parámetro *FROM* y mediante el parámetro *COPY*, se indica la ubicación del archivo jar de la aplicación. Además se especifica el directorio de trabajo de la aplicación con el parámetro *WORKDIR* y con el parámetro *ENTRYPOINT* se hace

[18] https://www.docker.com/products/docker-hub
[19] https://www.java.com/es/
[20] https://es.reactjs.org/
[21] https://spring.io

la llamada al jar como si se ejecutara en una consola (con *java – jar nombre.jar*). Por último se crea la imagen con el comando siguiente:

```
\textit{docker build -t "nombre de la imagen" -f "Dockerfile"}
```

- *React*: Para crear una imagen de un proyecto *React* en *docker*, ya con el proyecto compilado (utilizando pera ello el comando *npm run build*), se crea el archivo de configuración *dockerFile*. Antes de definir el *dockerFile* es necesario saber que para ejecutar un proyecto *React*, éste necesita servirse con un servidor de archivos estático. Esto se debe a que las aplicaciones de *React* usan enrutamiento del lado del cliente, donde las solicitudes web especificadas por URL(Localizador de Recursos Uniforme, del término anglosajón *Uniform Resource Locators*) son manejadas por el cliente, no pudiendo acceder al servidor para resolver las solicitudes. Para resolver este problema es necesario utilizar un servidor que sirva archivos estáticos, estos pueden ser *Nginx*[22], *Apache* [23] o *NodeJS* [24]. Ambos cuentan con una imagen en *Docker Hub* lo que facilita su utilización. En el proyecto se utiliza *Nginx* como servidor web, el cual sigue el estilo de *Apache* siendo orientado a eventos como *NodeJS*. Este servidor a diferencia del resto, permite el equilibrio de carga incluyendo las verificaciones de estado. Además mejora el consumo de CPU, actúa como un *proxy* inverso con almacenamiento en caché y como se dijo previamente, permite manejar archivos estáticos. Para construir el *dockerFile* del proyecto *React* utilizando como servidor *Nginx* se tiene que indicar en éste la versión del *Nginx* que se quiere utilizar mediante el parámetro *FROM* y con el parámetro *COPY* se agregan archivos del directorio actual del cliente *Docker*. Por último se crea la imagen con el comando visto anteriormente.

2.5.2. Kubernetes

Kubernetes es una plataforma de código abierto, portable y extensible, que se utiliza para automatizar y orquestar las implementaciones, el escalado y la administración de aplicaciones en contenedores, además de administrar cargas de trabajo y servicios [34]. La página oficial se puede encontrar aquí [25]. *Kubernetes* está ampliamente disponible y cuenta con un gran ecosistema, lo que le permite un crecimiento sólido.

Kubernetes se puede ver como una plataforma de contenedores, una plataforma de microservicios o una plataforma portable en la nube. Al igual que con *Docker*, se puede optimizar los flujos de trabajo de las aplicaciones para acelerar el tiempo de desarrollo. Además, permite a los usuarios poder escribir sus propios controladores e implementar todo tipo de módulos personalizados.

Los principales benéficos de usar esta plataforma residen en que agiliza la creación y despliegue de aplicaciones, un continuo desarrollo, integración y despliegue, separación de tareas, consistencia entre los entornos de desarrollo, pruebas y producción y portabilidad

[22] https://www.nginx.com/
[23] https://httpd.apache.org/
[24] https://nodejs.org/es/
[25] https://kubernetes.io/

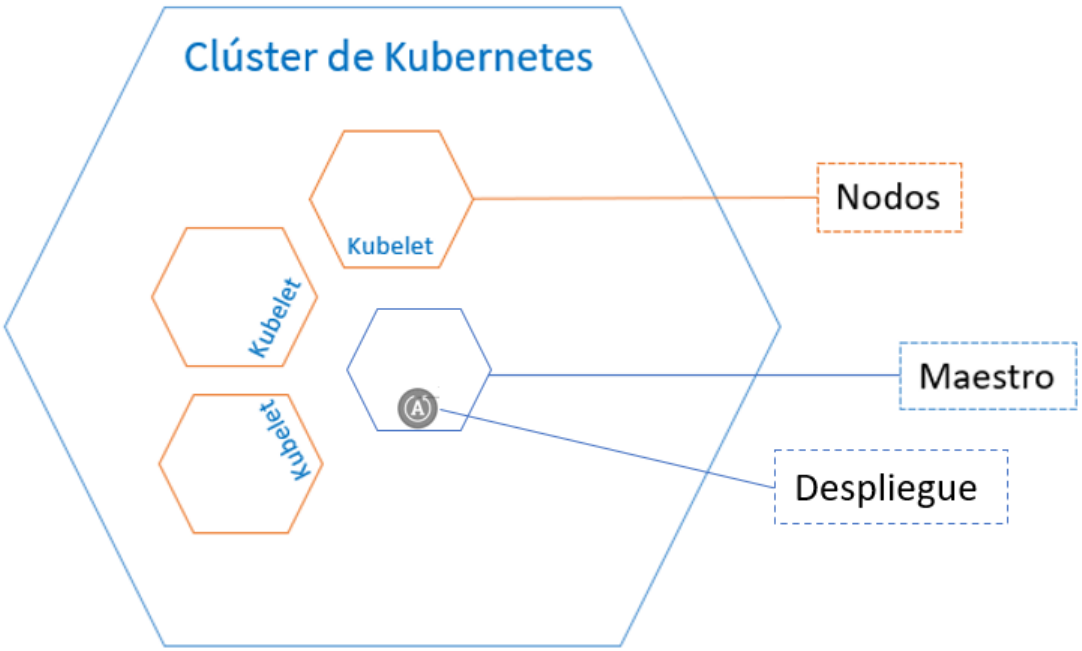

Figura 2.13: Clúster de *Kubernetes*.

entre nubes y distribuciones. Entre las limitaciones, se encuentra que no se pueden hacer despliegues de código fuente, ni compila aplicaciones, no es una plataforma como servicio (PaaS) y no provee servicios en la capa de aplicación.

Un clúster de *kubernetes* consta de dos tipos de recursos (que se refieren a máquinas, ya sean físicas o virtuales), tal y como se observa en la Figura 2.13 y se describen a continuación:

- Maestro o *Master*: Coordina todas las actividades del clúster, además de organizar (programar) aplicaciones, mantener el estado requerido de las aplicaciones, expandir e implementar actualizaciones, entre otras funciones. También recopila información de los nodos trabajadores y los *Pods* (la unidad de implementación más pequeña de *kubernetes* que contiene al menos un contenedor)

- Nodos: Son *workers* encargados de ejecutar la aplicación. Cada nodo contiene un agente llamado *Kubelet*, que gestiona el nodo y mantiene la comunicación con el maestro. Este nodo también tiene herramientas para manejar contenedores, como *Docker*.

Los *Pods* son una abstracción por encima de los contenedores. Estos pueden contener uno o más contenedores (aplicaciones), teniendo su propia IP dentro del clúster de *kubernetes*. La forma óptima de desplegar un contenedor (aplicación) es tener uno por *Pod*. Véase la Figura 2.14 donde se representa la estructura de un nodo.

Otros conceptos fundamentales para entender *kubernetes* son los siguientes:

- Despliegue: Es el encargado de supervisar el estado de los *Pods* y el número de réplicas de éste. se coloca en un nivel superior a los *Pods*.

- Servicio: Es la forma de exponer una aplicación que se ejecuta en un *pod* como un servicio de red. Con un servicio *kubernetes* le da a los *Pods* sus propias direcciones IP y un nombre DNS asociado.

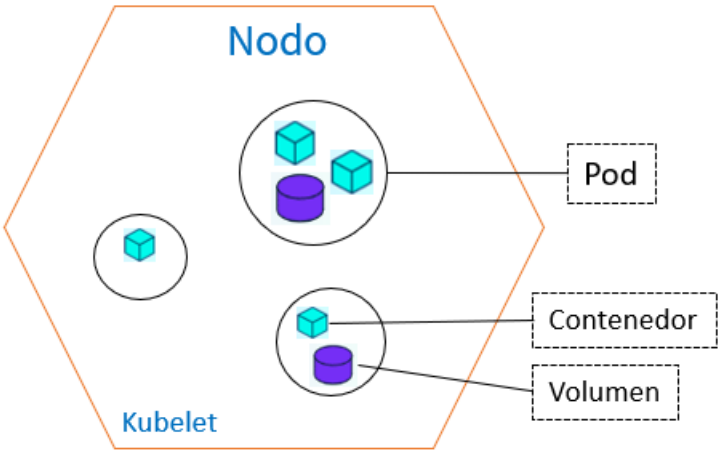

Figura 2.14: Nodo de *Kubernetes*.

- Kubelet: Agente de nodo que se ejecuta en cada nodo y lee los manifiestos del contenedor, y garantiza que los contenedores definidos estén iniciados y ejecutándose.
- kubectl: Herramienta de configuración de la línea de comandos de *kubernetes*.

Para crear un objeto en *kubernetes* mediante la API (bien de forma directa o usando *kubectl*), la petición a la API tiene que incluir toda la información en formato JSON en el cuerpo de la petición. Lo habitual es proporcionar la información a *kubectl* como un archivo *.yaml*. *kubectl* convierte esa información a JSON cuando realiza la llamada a la API.

El entorno de contenedores de *kubernetes* proporciona una serie de recursos importantes para los contenedores, proporcionando un sistema de archivos, que es una combinación de una imagen y uno o más volúmenes, información sobre el contenedor e información sobre otros objetos del clúster

Para poder organizar todos los pods desplegamos en *kubernetes* se tienen dos opciones: las etiquetas o *labels* y los nombres de espacio o *namespaces*. Las etiquetas sirven para dos cosas, por un lado, para poder identificar diferentes grupos de contenedores y para conectar diferentes tipos de recursos del clúster (*pods*, servicios, etc.). Por otro lado, los nombres de espacio son formas de agrupar los recursos y están diseñados para evitar una posible superposición entre etiquetas.

En este proyecto se agrupan todos los componentes bajo un mismo espacio o *namespaces*. Además, cada componente se ejecutará en un *pod* utilizando los servicios para tener su propia dirección IP y nombre DNS. Para el despliegue de los componentes se utilizarán las imágenes alojadas en *Docker*.

Capítulo 3

Diseño del sistema

En este capítulo, se realiza una descripción a alto nivel del sistema diseñado, para en el siguiente capítulo proporcionar detalles relativos a la implementación. Así, en primer lugar se identificarán los distintos componentes del sistema. A continuación, se describirá la estructura del modelo de datos diseñado.

3.1. Componentes del sistema

El sistema diseñado cuenta con una serie de componentes para ser desplegados en un sistema servidor (situado en el *fog* o el *cloud* del IoT) y un sistema sensor (situado en el *edge* del IoT). La interacción entre los componentes del sistema se muestra en la Figura(3.1). Estos componentes se definen como sigue:

- **Web front-end:** Se encarga de proporcionar una interfaz basada en web, donde los usuarios, acorde a sus privilegios, podrán visualizar los datos recolectados de CO_2 por el sistema y gestionar las configuraciones de los distintos sensores desplegados. Internamente este componente, que se desplegará en el sistema servidor, se divide en un cliente y un servidor web. El cliente transformará las solicitudes de los usuarios en peticiones que serán enviadas al servidor web mediante REST/HTTP. El servidor web responderá a las peticiones recibidas, llevando a cabo la acción solicitada. Cada uno de estos dos componentes internos (cliente y servidor web) se ejecutará en un *pod* de *Kubernetes*, utilizando *Nginx* como balanceador de carga.

- **Base de datos:** Se encarga de almacenar y gestionar la información obtenida por el sistema acerca de las mediciones de CO_2, así como información administrativa acerca de los usuarios del sistema, los sensores desplegados y sus ubicaciones. Esta información se almacenará en forma de documentos en formato JSON bajo *MongoDB*. Este componente, que se desplegará en el sistema servidor, se ejecutará sobre un *pod* de *Kubernetes*.

- **Controlador:** Se encarga de recoger y almacenar en la base de datos la información recolectada periódicamente por los distintos sensores del sistema en el *Broker* MQTT. La comunicación entre este componente y el *Broker* MQTT es a través del protocolo MQTT. Este componente, que se desplegará en el sistema servidor, será ejecutado en un pod de *Kubernetes*.

- **Broker MQTT:** Implementa el servicio de mensajería (haciendo uso de *Eclipse Mosquitto*) donde los publicadores (los sensores) publicarán información de la medición

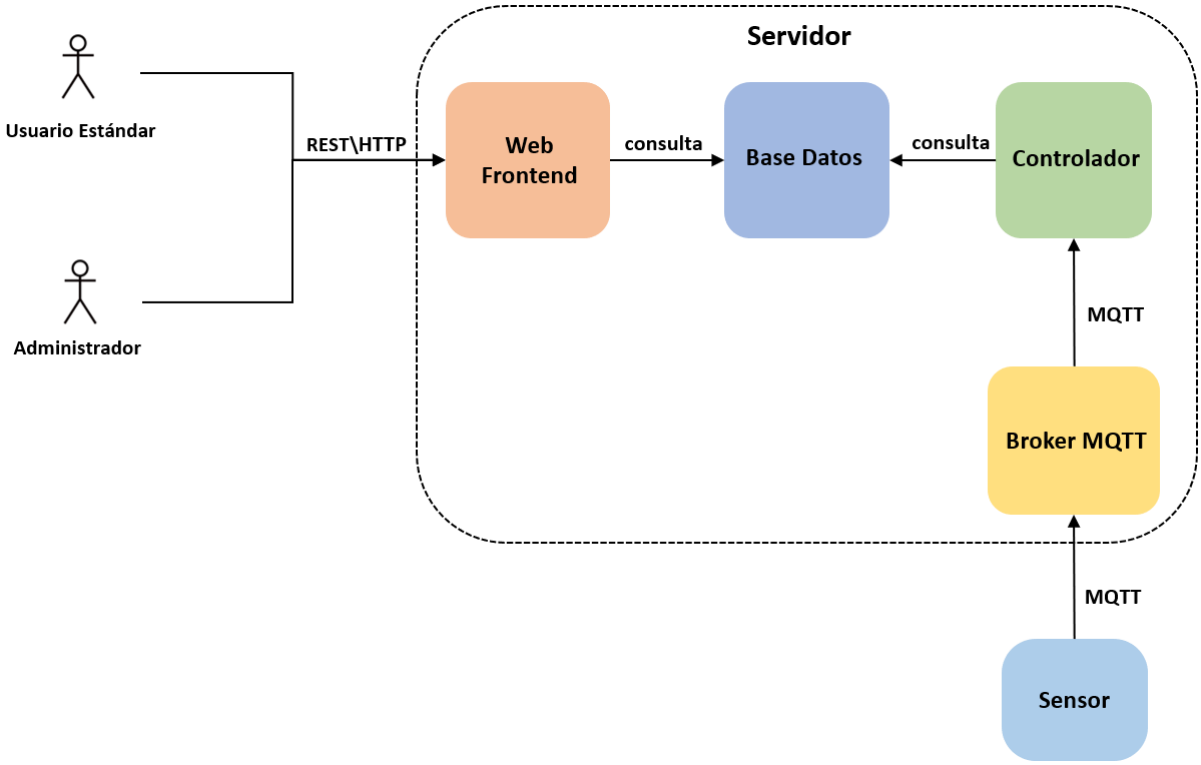

Figura 3.1: Interacción entre los distintos componentes del sistema.

de CO_2 bajo un tópico concreto (que hará relación a la ubicación física del sensor). Este componente, que se desplegará en el sistema servidor, se ejecutará sobre un *pod* de *Kubernetes*.

- **Sensor:** Se encarga de proporcionar mediciones periódicas acerca de la concentración de CO_2, usando para ello la plataforma hardware NodeMCU (en su versión ESP8266MOD-12F) junto al sensor MQ-135 (en la placa *AZ-Delivery* MQ-135), ambos conectados físicamente. Este componente, usando el protocolo MQTT, publicará la medición realizada en el *Broker* MQTT vía WiFi. Tal y como se desprende de esta descripción, este componente se desplegará en el *firmware* del sistema sensor.

3.2. Estructura del modelo de datos

En este apartado se describe la estructura lógica de la base de datos del sistema, incluyendo las relaciones y limitaciones que determinan la forma de almacenar los datos y de cómo se accede a ellos.

Tal y como se ha introducido previamente, el diseño considera una base de datos NoSQL, **MongoDB**, que se basa en el almacenamiento de documentos (en formato JSON) en distintas colecciones. En la estructura definida, la base de datos del sistema se denomina **db_sistema** y consta de cuatro colecciones llamadas **Usuario**, **Espacio**, **Sensor** y **SensorDatos**. Estas colecciones tienen la siguiente utilidad:

- **Usuario:** Almacena los datos de los usuarios que podrán acceder al sistema. Los usuarios serán añadidos por un usuario con permisos de administrador.

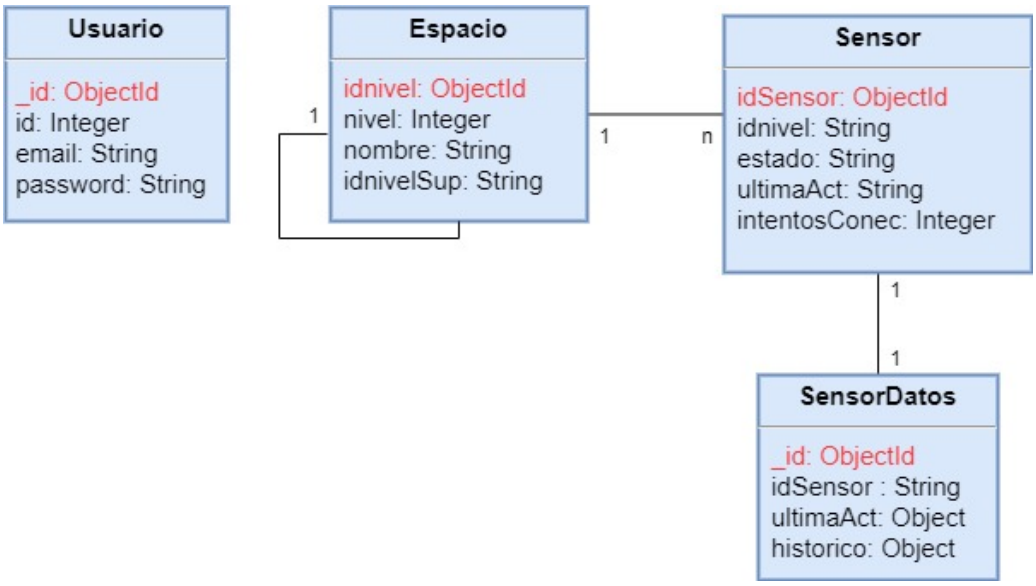

Figura 3.2: Modelo de datos.

- **Espacio:** Recoge los espacios o ubicaciones físicas donde se encuentran desplegados los sensores. Estos espacios serán gestionados por un usuario con permisos de administrador desde la interfaz de usuario (*Web front-end*).
- **Sensor:** Contiene los datos acerca de los sensores en el sistema. Los sensores serán añadidos por un usuario con permisos de administrador.
- **SensorDatos:** Almacena los datos recogidos por cada uno de los sensores en relación con la concentración de CO_2. Los usuarios tan solo pueden acceder a esta colección para visualizar los datos mediante la interfaz.

La relación lógica entre las distintas colecciones se recoge en la Figura 3.2, donde:

- **Usuario:** No tiene relación con ninguna otra colección de la base de datos.
- **Espacio y Sensor:** Un Espacio puede tener múltiples Sensores asociados, pero un Sensor solo puede estar desplegado en un Espacio. **Espacio y Espacio:** Un Espacio puede tener asociado otro Espacio que será jerárquicamente superior. Por ejemplo, un edificio y cada una de sus plantas.
- **Sensor y SensorDatos:** Un Sensor solo puede tener asociado un SensorDatos, al igual que un SensorDatos solo puede tener un Sensor. Tal y como se verás más adelante, un documento de la colección SensorDatos permite almacenar el histórico de datos capturados por un Sensor.

Los documentos, para cada una de las colecciones descritas previamente, contienen los siguientes campos:

- **Usuario**

```
Usuario : {
    _id : ObjectId,
```

```
    id : Integer,
    email : String,
    password : String
}
```

- **_id:** Tipo *ObjectId*. Identifica de forma única el usuario. Este dato es autogenerado por el sistema al insertar un nuevo usuario. Nótese que en *MongoDB* el tipo *ObjectId* consiste en una representación compuesta por 12 bytes, donde 4 bytes lo conforma un valor *timestamp* en relación al momento de creación del documento, 5 bytes un número aleatorio y 3 bytes para un contador autoincremental que parte de un valor inicial aleatorio. Este valor de 12 bytes es convertible fácilmente en una representación *String* para recoger las relaciones lógicas entre las colecciones.
- **id:** Tipo *Integer*. Identifica el tipo de usuario, tomando un valor 1 si se trata de un usuario administrador y 2 en caso de tratarse de un usuario estándar.
- **email:** Tipo *String*. Almacena el email del usuario, el cual servirá como nombre de usuario para el acceso a la interfaz y para identificar al usuario de forma única en el sistema.
- **password:** Tipo *String*. Almacena la contraseña del usuario con la que accederá a la interfaz.

- **Espacio**

```
Espacio : {
    idnivel : ObjectId,
    nivel : Integer,
    nombre : String,
    idnivelSup : String,
}
```

- **idnivel:** Tipo *ObjectId*. Identifica de forma única al espacio donde se desplegarán los sensores. Este identificador es autogenerado por el sistema al insertar un nuevo espacio.
- **nivel:** Tipo *Integer*. Representa el nivel jerárquico en el que se encuentra el espacio dentro de una posible infraestructura. Por ejemplo, se puede identificar en un primer nivel el nombre deseado para el conjunto de infraestructuras, en un segundo nivel pueden estar todas las infraestructuras(Nombre del edificio, local o centro), en un tercer nivel los pisos de la infraestructura y así hasta llegar al nivel donde se encuentra el espacio.
- **nombre:** Tipo *String*. Contiene el nombre del espacio o del nivel.
- **idnivelSup:** Tipo *String*. Contiene el identificador (*idnivel*) del Espacio que se encuentra en el nivel jerárquico inmediatamente superior al actual. Este valor hará referencia a un campo *idnivel* (de esta misma colección y con tipo *ObjectId*) convertido a *String*.

- **Sensor**

```
Sensor : {
    idSensor : ObjectId,
    idnivel : String,
    estado : String,
    ultimaAct : String,
    intentosConec : Integer,
}
```

- **idSensor:** Tipo *ObjectId*. Identifica de forma única el elemento sensor. Este identificador es autogenerado por el sistema al insertar un nuevo sensor.
- **idnivel:** Tipo *String*. Identifica de forma única el Espacio donde el sensor está desplegado. Este valor hará referencia a un campo *idnivel* (de la colección Espacios y con tipo *ObjectId*) convertido a *String*.
- **estado:** Tipo *String*. Indica el estado en el que se encuentra el sensor, estos son esperando (estado que tiene el sensor cuando se incorpora al sistema), ejecutando (estado que tiene el sensor cuando está ejecutado y proporcionando mediciones de CO_2) y error (estado que tiene el sensor cuando el servidor detecta error en el sensor físico). Un ejemplo del estado de error es que el sensor físico pase un tiempo determinado sin enviar datos al servidor.
- **ultimaAct:** Tipo *String*. Indica la fecha (hora incluida) de la última captura de información por parte del sensor acerca de la concentración de CO_2.
- **intentosConec:** Tipo *Integer*. Contiene el número de intentos de conexión del sensor en caso de caídas del sistema o no respuestas del sensor. En caso de superar un número máximo de intentos, el sensor pasará ha estado error para que el administrador verifique la comunicación con el elemento sensor.

- **SensorDatos**

```
SensorDatos : {
    _id : ObjectId,
    idSensor : String,
    estado : String,
    ultimaAct : Object,
    histórico : Object,
}
```

- **_id:** Tipo *ObjectId*. Identifica de forma única la información capturada por un sensor específico. Este identificador es autogenerado por el sistema al insertar un nuevo documento en esta colección.
- **idSensor:** Tipo *String*. Identifica de forma única el Sensor acerca del cual se está almacenando información. Este valor hará referencia a un campo *idSensor* (de la colección Sensor y con tipo *ObjectId*) convertido a *String*.
- **ultimaAct:** Tipo *Object*. Almacena información acerca de la última captura realizada, por lo que este campo se actualizará en cada llegada de una nueva captura por parte de un sensor. Concretamente, se guarda la fecha, la hora y el valor de concentración de CO_2 obtenido. Para ello, se utiliza un tipo propio (un documento incrustado) con el siguiente formato:

```
ultimaAct : {
    date :String ,
    time : String,
    dato : String,
}
```

- **histórico:** tipo *Object*. Almacena un listado de todas las mediciones realizadas por el sensor, por lo que este campo se actualizará en cada llegada de una nueva captura por parte de un sensor. Cada uno de los elementos de la lista sigue el mismo formato presentado para el campo *ultimaAct*. Para ello, se utiliza un tipo propio (un documento incrustado) con el siguiente formato:

```
histórico :
    [
      {
        date :String ,
        time : String,
        dato : String,
      },
      {
        date :String ,
        time : String,
        dato : String,
      },
      ...
    ]
```

En la Figura 3.3 se presenta un ejemplo de la estructura que podrían tomar los datos en *MongoDB* durante el funcionamiento del sistema. En este ejemplo, se puede ver como por cada colección se crea un documento, con la estructura descrita previamente. Además, en el caso del documento SensorDatos1 se representan los documentos incrustados en él.

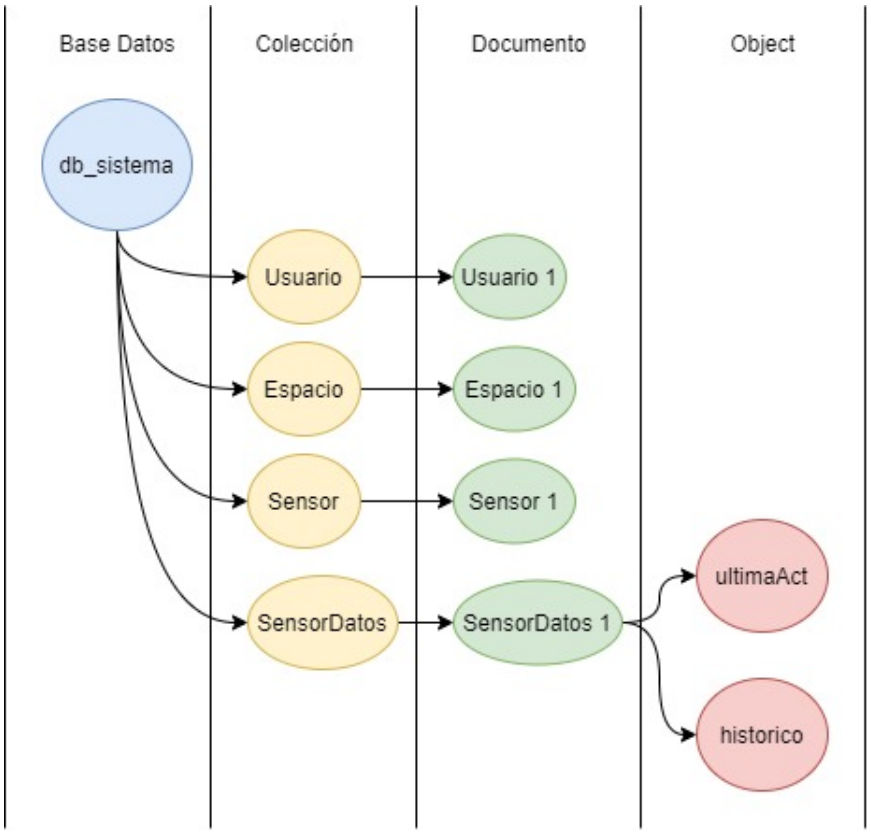

Figura 3.3: Ejemplo del modelo de datos durante la ejecución del sistema.

Capítulo 4

Implementación del sistema

En este capítulo, se proporcionan los detalles relativos a la implementación de cada uno de los componentes implicados en el diseño realizado en el capítulo previo. Estos componentes se encuentran representados en la Figura 4.1, la cual se basa en la Figura 3.1 pero con un mayor nivel de detalle, principalmente en relación a la tecnología utilizada y su división en subcomponentes. Así, se va a detallar el proceso de implementación de los siguientes componentes y subcomponentes: *Broker* MQTT, Base de Datos, Controlador, *Web Frontend* y Sensor.

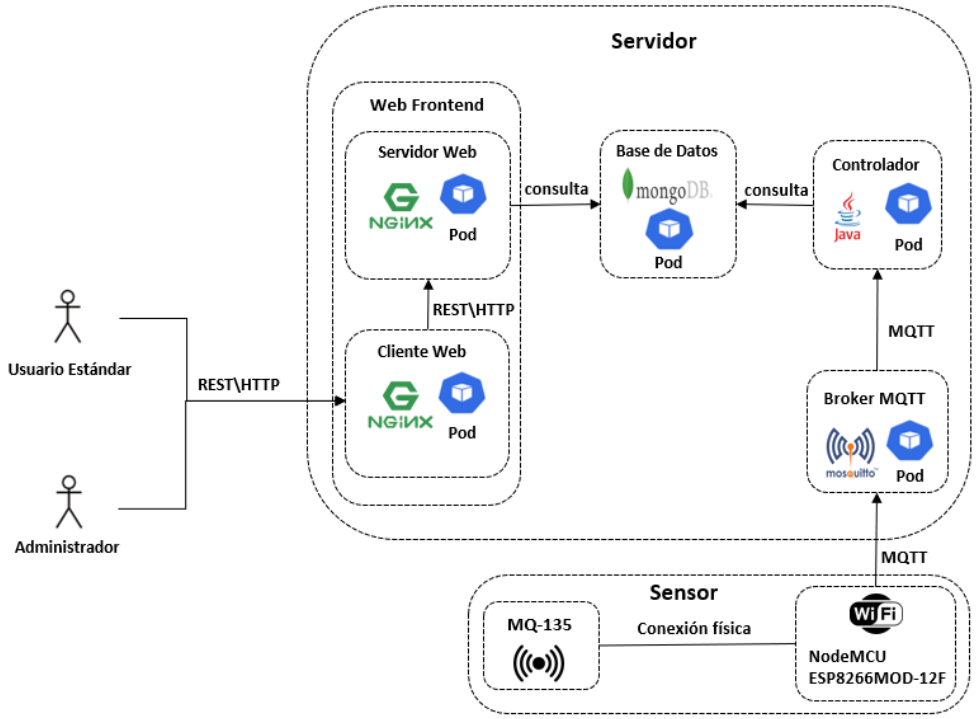

Figura 4.1: Interacción entre los distintos componentes del sistema con información ampliada acerca de tecnologías y subcomponentes.

4.1. Implementación del sistema servidor

4.1.1. Broker MQTT

Tal y como se ha descrito previamente, este componente ejecuta el *broker Eclipse Mosquitto* en un *pod* de *kubernetes* (véase la Figura 4.2). Puesto que *Eclipse Mosquitto* es una pieza de software libre con licencia (EPL/EDL licensed)[1], esta sección se centrará en explicar los yaml de despliegue de *Eclipse Mosquitto*. La versión utilizada de este *broker* es la última disponible (2.0.8)[2][3].

Figura 4.2: Componente Broker MQTT.

Los Yaml necesarios para el despliegue se encuentran en la ruta *../Kubernetes/mosquitto* de la raíz del proyecto. Estos ya fueron definidos en *Docker Hub* con la intención de que el *Broker MQTT* funcione e interactué correctamente con los módulos del sistema, por lo que solo sería necesario desplegarlos en el servidor, sin necesidad de definirlos o modificarlos. Los Yaml son los siguientes:

- *mosquitto-config.yaml*
- *mosquitto-deployment.yaml*
- *mosquitto-service.yaml*

El contenido de los Yaml de despliegue del *broker* se describe a continuación:

1. *mosquitto-config.yaml*: Contiene un *ConfigMaps*(objeto de la API utilizado como variables de entorno, argumentos de la línea de comandos o como ficheros de configuración) de *kubernetes*. Este fichero contiene las configuraciones que necesita *mosquitto* para funcionar correctamente. Estas son el establecimiento del puerto 1883 como escuchador por defecto de *mosquitto* y se define que éste no tendrá ningún mecanismo de seguridad de los que dispone *mosquitto*, según lo visto en la Sección 2.3.1.2.

2. *mosquitto-service.yaml*: Expone *Mosquitto* como un servicio de red en *kubernetes*. La Figura 4.3) muestra una sección de código de este fichero donde se expone el servicio *Mosquitto* en el puerto 30210 para la comunicación desde fuera del servidor y la comunicación interna con el componente controlador del servidor (usando el tipo de servicio *NodePort* para ello). Además, el puerto 30210 con servicio *NodePort* informa al puerto asignado por *targetPort* que es el 1883 de los datos que escucha. El puerto 1883 es el que utiliza por defecto *Mosquitto*.

[1] https://mosquitto.org/
[2] https://mosquitto.org/blog/2021/02/version-2-0-8-released/
[3] https://hub.docker.com/_/eclipse-mosquitto

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mosquitto-service
  namespace: sistematfg
  labels:
    name: mosquitto
spec:
  ports:
    - protocol: TCP
      port: 1883
      targetPort: 1883
      nodePort: 30210
  selector:
    app: mosquitto
  type: NodePort
```

Figura 4.3: Sección de código en *mosquitto-config.yaml* que permite el despliegue de *Mosquitto* como servicio en *Kubernetes*.

3. *mosquitto-deployment.yaml*: Permite ejecutar *Mosquitto* en un *pod* de *Kubernetes*. Este *pod* está definido del tipo *StatefulSet*, pues permitirá que, en caso de caída del *pod*, se pueda recuperar manteniendo su estado anterior, lo que pasa por conservar su nombre de *pod*, IP y DNS. Como se observa en la sección de código del *mosquitto-deployment.yaml* mostrada en la Figura(4.4), el *StatefulSets* utiliza el *Service* y el *ConfigMaps* mencionados anteriormente.

Con los ficheros descritos anteriormente desplegados (el despliegue se explicará en la Sección B.1, dentro del manual de usuario), el *broker* es accesible desde fuera del servidor conociendo la dirección IP del servidor y el puerto del componente expuesto al exterior. La comunicación con este componente desde el exterior del servidor la realizan los sensores en base al protocolo MQTT, conociendo la dirección IP fija del servidor y el correspondiente puerto del componente (30210). La comunicación con este componente desde el interior del servidor la realiza el componente Controlador según el protocolo MQTT. Como estos dos componentes se ejecutan cada uno en un *pod* de *kubernetes* dentro de un mismo clúster, el componente Controlador accederá localmente al *pod* que implementa el *broker* conociendo su puerto (30210).

4.1.2. Base de Datos

Este componente ejecuta la base de datos *MongoDB* en un *pod* de *kubernetes* (véase la Figura 4.5). Como *MongoDB* es una pieza de software libre con licencia (GNU AGPL v3.0)[4], esta sección se centrará en explicar los yaml de despliegue de *MongoDB* en el servidor. La versión utilizada de esta base de datos es la última disponible (v4.4.6)[5].

Los Yaml necesarios para el despliegue se encuentran en la ruta (../*Kubernetes*/*mongobd*) de la raíz del proyecto. Estos ya fueron definidos en *Docker Hub* con la intención de que la

[4] https://www.mongodb.com/
[5] https://docs.mongodb.com/manual/release-notes/4.4/

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mosquitto-d
  namespace: sistematfg
spec:
  selector:
    matchLabels:
      app: mosquitto
  serviceName: mosquitto-service
  replicas: 1
  template:
    metadata:
      labels:
        app: mosquitto
        replicaset: MainRepSet
    spec:
      containers:
        - name: mosquitto-container
          image: eclipse-mosquitto:2.0.8
          ports:
            - containerPort: 1883
          # resources:
          volumeMounts:
            - name: mosquitto-config
              mountPath: /mosquitto/config/mosquitto.conf
              subPath: mosquitto.conf
      volumes:
        - name: mosquitto-config
          configMap:
            name: mosquitto-config
```

Figura 4.4: Sección de código en *mosquitto-deployment.yaml* que permite ejecutar *Mosquitto* en un *pod* de *Kubernetes*.

base de datos funcione e interactué correctamente con los módulos del sistema, por lo que solo sería necesario desplegarlos en el servidor, sin necesidad de definirlos o modificarlos. Los Yaml son los siguientes:

- mongodb-service.yaml
- mongodb-statefulset.yaml

El contenido de los Yaml de despliegue de *MongoDB* se describe a continuación:

1. *mongodb-service.yaml*: Expone *MongoDB* como un servicio de red en *kubernetes*. La Figura 4.6 muestra una sección de código de este fichero, donde *MongoDB* se expone como servicio para comunicación interna en el puerto 27017, siendo éste un valor por defecto. Este puerto solo es conocido dentro del clúster de *kubernetes*.

2. *mongodb-statefulset.yaml*: Permite ejecutar *MongoDB* en un *pod* de *Kubernetes* de tipo *Statefulset*. El *StatefulSets* utiliza el *Service* mencionado en el paso anterior para exponer *Mongodb*. Las configuraciones más importantes de este fichero son las siguientes:

    - Uso de un *pod* de tipo *StatefulSet*.que permitirá que, en caso de que el *pod* se caiga, se pueda volver a recuperar manteniendo su estado anterior, lo que pasa por conservar su nombre de *pod*, IP y DNS. Además de permitir desplegar

Figura 4.5: Componente Base de datos.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mongodb-s
  namespace: sistematfg
  labels:
    name: mongodb
spec:
  ports:
  - port: 27017
    targetPort: 27017
  clusterIP: None
  selector:
    role: mongodb
```

Figura 4.6: Sección de código en *mongodb-service.yaml* que expone *MongoDB* como un servicio de red en *Kubernetes*

aplicaciones con persistencia, algo prácticamente imprescindible en el caso de una base de datos.

- Se establece tres *Replica sets* de Mongo. De esta manera se podrá desplegar *MongoDB* en configuración *Replica sets* vista en el apartado de base teórica. Así, se tendrá un nodo configurado como maestro y los otros dos como nodos secundarios. Además, este fichero de configuración permite realizar de manera automatizada (bajo la etiqueta *commands*) el establecimiento de las tres réplicas en un *Replica Set* denominado *MainRepSet*. Véase la Figura 4.8.
- Se establece la persistencia de los datos con la asignación de un *PersistentVolume* (o volumen persistente, usado para administrar el almacenamiento duradero en un clúster de *Kubernetes*) y un *PersistentVolumeClaim* (similares a un *Pod*, permitiéndole consumir recursos de almacenamiento *PersistentVolume*). Para la asignación se utiliza la etiqueta *volumeClaimTemplates* (lista de afirmaciones a las que los *pods* pueden hacer referencia) que permite generar de forma automática un objeto *PersistentVolumeClaim* que define y aprovisiona de manera dinámica un *PersistentVolume*, para que pueda ser utilizado por el *pod* como almacenamiento. Como se puede ver en la Figura(4.7), con la etiqueta *volumeClaimTemplates* se definen las siguientes características del *PersistentVolumeClaim*:
  - Nombre del *PersistentVolumeClaim* igual a *mongodb-storage*.
  - *accessModes*(descriptor de las capacidades del volumen) igual a *ReadWriteOnce*, que permite que el *PersistentVolumeClaim* se puede montar como

```
volumeClaimTemplates:
- metadata:
    name: mongodb-storage
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

Figura 4.7: Sección de código en *mongodb-statefulset.yaml que configura el almacenamiento en MongoDB.*

```
containers:
  - name: mongod-container
    image: mongo
    command:
      - "mongod"
      - "--bind_ip"
      - "0.0.0.0"
      - "--replSet"
      - "MainRepSet/mongodb-d-0.mongodb-s.sistematfg.svc.cluster.local:27017,
        mongodb-d-1.mongodb-s.sistematfg.svc.cluster.local:27017,
        mongodb-d-2.mongodb-s.sistematfg.svc.cluster.local:27017"
    ports:
      - containerPort: 27017
    volumeMounts:
      - name: mongodb-storage
        mountPath: /data/db
```

Figura 4.8: Sección de código en *mongodb-statefulset.yaml que configura el contenedor de MongoDB.*

lectura-escritura.

- *storage*(almacenamiento) de un GB para el *PersistentVolumeClaim*.

Con los ficheros descritos anteriormente desplegados (el despliegue se explicará en la Sección B.1, dentro del manual de usuario), la base de datos es accesible internamente al servidor por los componentes *Web Front-end* y el Controlador. Ambos componentes realizarán consultas a la base de datos, utilizando para ello la cadena definida en la configuración del contenedor en *mongodb-statefulset.yaml*. Esta comunicación se establecerá según el protocolo definido para MongoDB[6], que determina los términos requeridos para una comunicación consistente.

4.1.3. Controlador

Este componente se encarga de recoger los mensajes en el *Broker* MQTT y guardarlos en la base de datos. Para ello, por cada elemento sensor desplegado, se creará un hilo que escuchará e identificará nuevos mensajes presentes en el *broker* acerca del *topic* específico de este sensor. Una vez detectados, se almacenarán en la base de datos. Este componente también se encarga de administrar las conexiones con el *broker*, para que en el caso de

[6] https://docs.mongodb.com/manual/reference/mongodb-wire-protocol/

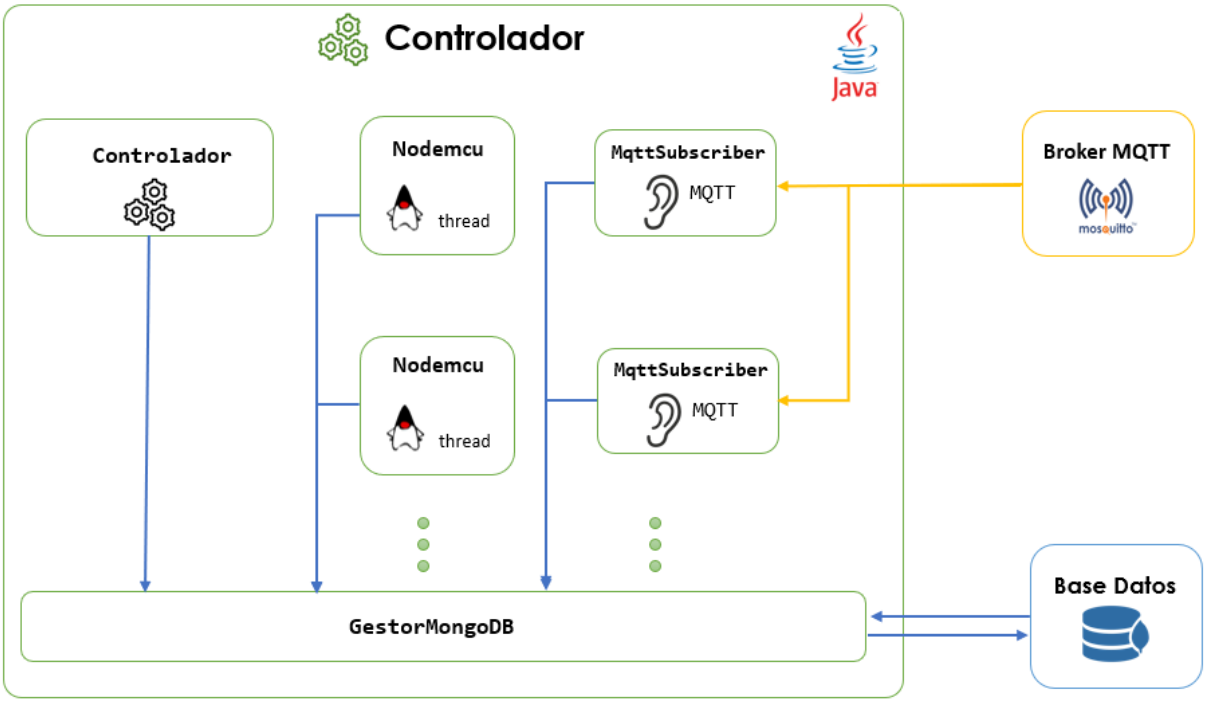

Figura 4.9: Diagrama lógico funcional del componente Controlador.

```
public MqttSubscriber(String brokerUrl, String clientId, String topic,  GestorMongoDB newConect) {
    this.brokerUrl = brokerUrl;
    this.clientId = clientId;
    this.topic = topic;
    this.newConect = newConect;
}
```

Figura 4.10: Constructor de la clase *MqttSubscriber*.

perdida de la conexión, se lance un nuevo hilo deteniendo el anterior.

Puesto que este componente es un desarrollo específico de este proyecto se tiene que generar el contenedor de forma manual mediante el *framework Spring*[7], en su versión 5.2.8. Este componente sigue el diagrama lógico funcional mostrado en la Figura 4.9, el cual está compuesto por las siguientes clases:

- **MqttSubscriber:** Clase que está escuchando a la espera de la llegada de mensajes al *broker* con un *topic* relacionado con un sensor en concreto. Tan pronto como identifica un nuevo mensaje, lo guarda en la base de datos. Esta clase utiliza la biblioteca *Paho* [8] de Java, la cual implementa un cliente MQTT.

  El constructor de la clase se puede ver en la Figura 4.10, donde los parámetros involucrados los completará la clase *nodemcu*, que son el *brokerUrl* (la dirección del *broker* MQTT), *clientID* (el identificador de cliente generado aleatoriamente), *topic* (el tópico asociado a un sensor concreto) y *newConect* (una instancia de la clase *GestorMongoDB*).

  Los métodos de la clase *MqttSubscriber* son los siguientes:

[7] https://spring.io
[8] https://www.eclipse.org/paho/

- **subscribe():** Método que permite iniciar el proceso de escucha de un *topic* concreto en el *broker*.
- **messageArrived():** Método que se ejecuta cuando llega un mensaje procedente del *broker*. Al recibir el mensaje lo encapsula en un JSON y lo guarda en la base de datos, añadiendo la hora y la fecha en el que fue recibido. EL JSON tiene la siguiente estructura:

```
{   "date": LocalDate.now().toString()
    "time": LocalTime.now().toString()
    "dato": json\_data.get("dato").textValue()
}
```

  donde *LocalDate.now().toString()* se utiliza para obtener la fecha actual del servidor, *LocalTime.now().toString()* para obtener la hora y *json_data.get("dato").textValue()* para extraer la medida de concetración de CO_2 del mensaje recibido del *broker*. Este JSON generado se almacena en la colección *SensorDatos* de la base de datos, actualizando el documento *ultimaAct* y añadiendo un *Object* nuevo al documento histórico.
- **stopCOnection():** Método que permite detener la ejecución del proceso de escucha, primero deteniendo la conexión con el *broker* y luego cerrándola.

- **GestorMongoDB:** Clase que contiene todos los métodos necesarios para realizar las consultas a la base de Datos. Los métodos van desde establecer la conexión con la base de Datos hasta las consultas de insertar, actualizar o eliminar un documento o valor de éste. Esta clase sirve de intermediaria de las otras clases del controlador con la base de datos, para ello importa el paquete *com.mongodb.client*.

- **Nodemcu:** Clase que se relaciona con los sensores del sistema. Al crearse una instancia de ésta se le especifica los datos del sensor con el que está relacionado. Esta clase se encarga de crear y lanzar una instancia de la clase **MqttSubscriber** para que ésta recoja los mensajes en el *Broker* MQTT del sensor que referencia. Además comprueba periódicamente la validez del Sensor, comprobando si éste está enviado datos al *broker* o si el usuario administrador lo ha eliminado del sistema.

  El constructor de la clase se puede ver en la Figura 4.11, donde los parámetros involucrados los completará la clase *Controlador* al crear una instancia. Estos parámetros son *id*(el identificador único asociado con el sensor que se desea monitorizar), *topic* (el tópico asociado al sensor), *brokerUrl* (la dirección del *broker* MQTT) y *newConect* (una instancia de la clase *GestorMongoDB*).

```
public Nodemcu(String id, String topic, String brokerUrl, GestorMongoDB newConect) {
    this.id = id;
    this.topic = topic;
    this.brokerUrl = brokerUrl;
    this.newConect = newConect;
}
```

Figura 4.11: Constructor de la clase *Nodemcu*.

Los métodos de la clase *Nodemcu* son los siguientes:

- **comprobarEstado():** Comprueba en la base de datos la existencia del sensor que se monitoriza, si fue eliminado entonces el método devuelve *true*, y en caso contrario *false*.

- **comprobarEstadoComu():** Comprueba el estado de la comunicación entre el sensor y el sistema. Para ello, consulta la última entrada recogida en la base de datos para el sensor. Si la última actualización es del mismo día y hora y con un margen de 6 minutos, entonces devuelve *true*, en caso contrario retorna *false*.
- **comprobarContComu():** Comprueba si el contador de reinicios (el cual se incrementa cuando no se reciben datos del sensor en los últimos 6 minutos) es mayor que 5. Si supera este umbral, entonces devuelve *true* y en caso contrario *false*.
- **run():** Método que permite iniciar el hilo (mediante la llamada al método *start()* desde la clase Controlador) que terminará automáticamente cuando el sensor sea eliminado del sistema. Este procedimiento sigue el diagrama de flujo en la Figura 4.12. En este diagrama, en primer lugar se crea una instancia de la clase *MqttSubscriber* vista antes y se actualiza en la base de datos el estado en ejecución del sensor (colección *Sensor* y campo *estado* asociado al documento relativo al sensor). A continuación, comienza un bucle que termina solo si es eliminado el sensor del sistema. Dentro del bucle, se esperan 25 segundos y se comprueba si en el sensor fue eliminado (*comprobarEstado()*), parando la ejecución en caso afirmativo. A continuación, se verifica el estado de la comunicación (*comprobarEstadoComu()*) y se incrementa el número de reinicios (en la colección *Sensor* y campo *intentosConec* asociado al documento relativo al sensor) en caso de detectar un problema con la comunicación. En tal caso problemático, si se alcanza el valor umbral (*comprobarContComu()*) se finaliza el proceso y se actualiza la base de datos el estado del sensor. En caso de que no hubiera problemas con la comunicación, entonces se reinicializa el contador de intentos de conexión del sensor.

- **Controlador:** Es la clase principal del componente, por ello, el uso del mismo nombre para su definición. Esta clase se encarga de crear y lanzar las instancias requeridas de la clase *Nodemcu*, tantas como sensores en el sistema. Esta clase no tiene constructor. Los métodos de la clase *Controlador* son:

  - **crearEspacio():** Crea una instancia de la clase *Nodemcu* con el *id* y el *topic* pasado como parámetro. A continuación, mediante la función *start()*, se llama internamente al método *run()* de la clase *Nodemcu* y ejecuta su contenido (véase la Figura 4.13).
  - **getNewNodes():** Se encarga de crear un *HashMap*(implementación basada en tablas *hash* de la interfaz *Map* de *Java*, donde un *Map* es una colección de pares clave-valor) de todos los sensores creados por el usuario administrador que se quieren monitorizar. Para ello, se realiza una consulta en la base de datos (en la colección *Sensor* y campo *estado*) acerca de los sensores que tienen como estado escuchando (estado que indica que fue creado, pero que no se está ejecutando). El *HashMap* tiene como clave el *id* del sensor y de valor del *topic*.
  - **checkNodesUpdate():** Se encarga de verificar la última actualización del sensor en la base de datos (en la colección *Sensor* y campo *ultimaAct*). Si la última actualización es menor a dos minutos, entonces se presupone que está caído el hilo y lo crea nuevamente.

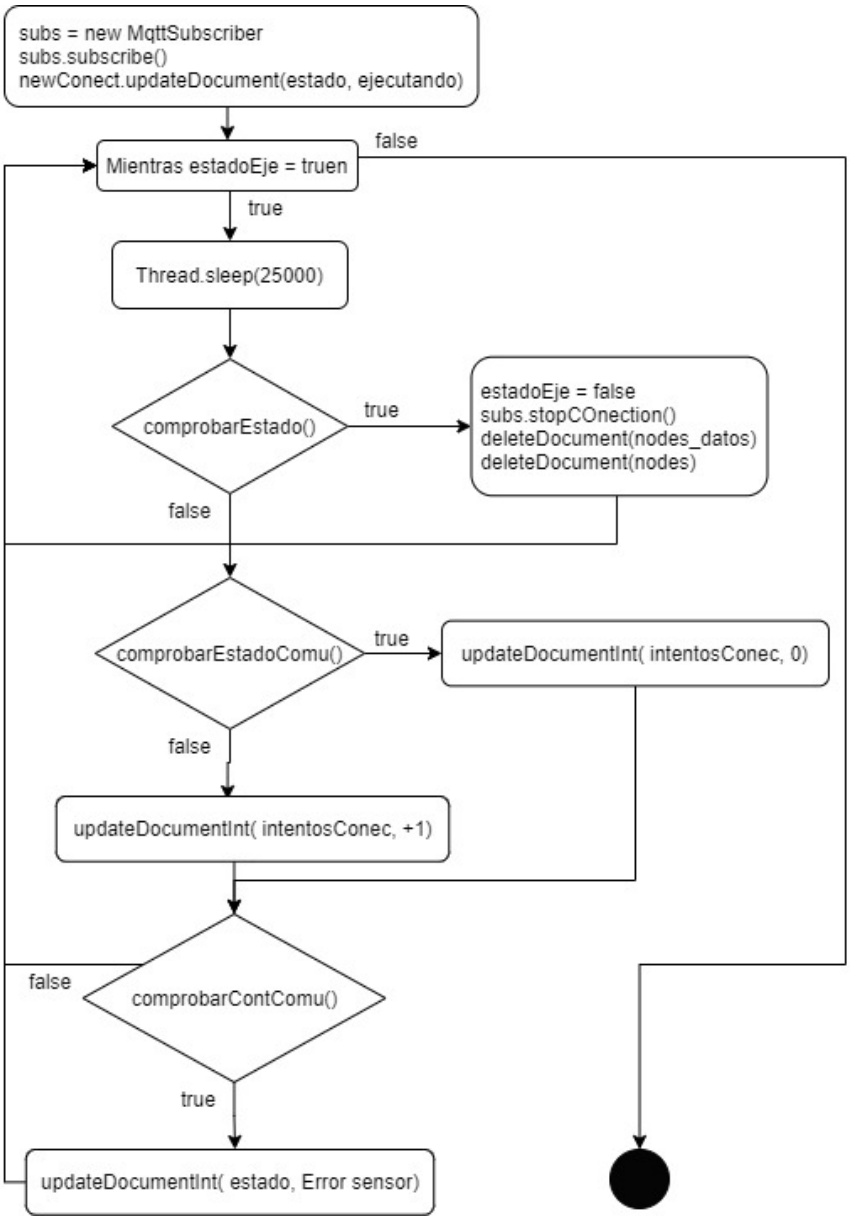

Figura 4.12: Diagrama de flujo del método *run()* de la clase *Nodemcu*.

- **run():** Método que se ejecuta al iniciar el componente *Controlador*, siguiendo el diagrama de flujo en la Figura 4.14. Así, en primer lugar se crea una instancia de la clase *GestorMongoDB* que se encarga de establecer la conexión con el módulo de la base de datos. Esta instancia se utilizará en las clases *MqttSubscriber*, *Nodemcu* y el propio Controlador para interactuar con la base de datos. Después de establecer la conexión con la base de datos, se inicia un bucle infinito que se mantiene durante toda la ejecución del método. En este bucle, utilizando el método *getNewNodes()*, se genera una lista de nodos a monitorizar. En el caso que la lista tenga elementos, se generan tantas instancias de la clase *Nodemcu* como elementos en la lista (*crearEspacio()*). En el caso de que la lista esté vacía, se comprueba la última actualización de los sensores (*checkNodesUpdate()*). Por último, se esperan 10 segundos antes de comenzar la siguiente iteración.

```
public void crearEspacio(String id, String topic) {
    Nodemcu nuevonode = new Nodemcu(id, topic, brokerUrl, newConect);
    nuevonode.start();
}
```

Figura 4.13: Función *crearEspacio()* de la clase *Controlador*.

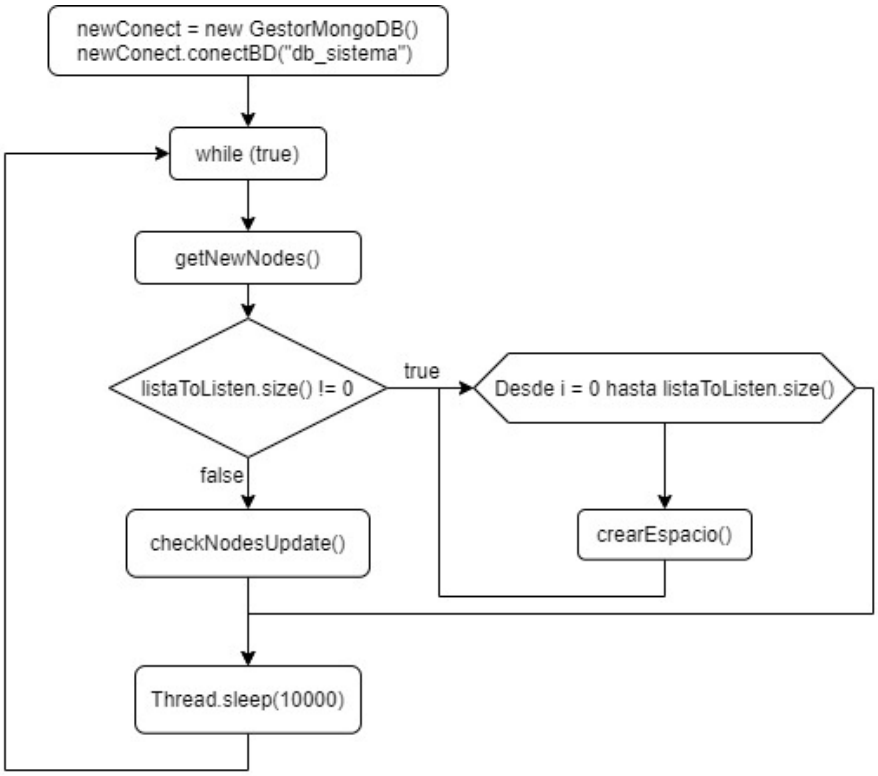

Figura 4.14: Diagrama de flujo del método *run()* de la clase *Controlador*.

4.1.4. Web Frontend

Este componente se compone de dos aplicaciones: un cliente y un servidor web. Cada una de estas aplicaciones se ha desarrollado utilizando *Reactjs*[9], una biblioteca de *JavaScript* para construir interfaces de usuario. Cada aplicación se ejecuta en un *pod* de *Kubernetes*, utilizando *Nginx* como balanceador de carga. En la Figura 4.15 se detalla la interacción de este componente con el exterior (usuarios) y con otro componente del servidor (la base de datos). A continuación, se proporcionan los detalles de implementación para cada una de estas aplicaciones.

4.1.4.1. Servidor web

Esta aplicación se encarga de recibir y responder a las solicitudes enviadas por el cliente web. Las solicitudes las responde haciendo uso de la base de datos a través de consultas a ésta. La aplicación se ejecuta en un *pod* de *Kubernetes*, la cual expone un puerto que será conocido por el cliente web y que permite la conexión únicamente a los componentes del clúster (los componentes internos del servidor).

[9] https://es.reactjs.org/

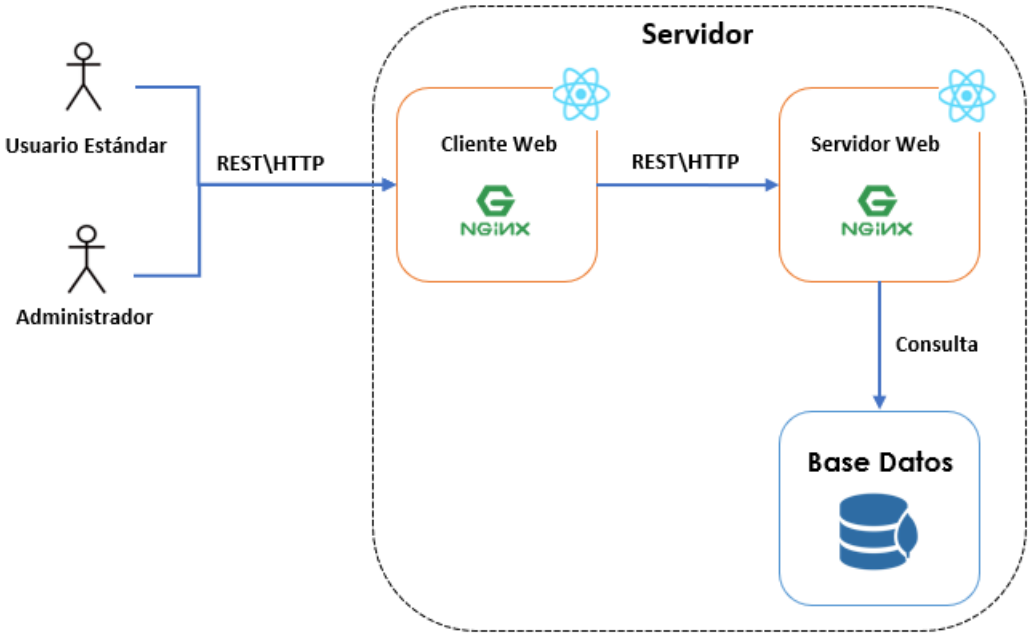

Figura 4.15: Interacción del componente Web Frontend con otros componentes del servidor y con los usuarios del sistema.

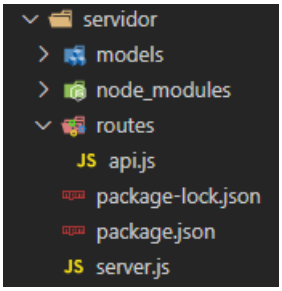

Figura 4.16: Estructura de ficheros del servidor web.

La comunicación del cliente con el servidor se realiza a través de la *API REST* (acrónimo del término anglosajón *representational state transfer* [35]), donde las solicitudes se entregan por medio de *HTTP* en formato *JSON*. Para la comunicación del servidor web con la base de datos se utiliza *mongoose*. Tal y como se ha introducido previamente, el servidor web fue programado con *Reactjs* utilizando las siguientes bibliotecas:

- *Expressjs*[10]: *Framework* web alojado dentro del entorno de ejecución *NodeJS*, que proporciona un conjunto sólido de características para las aplicaciones web, además es robusto, rápido y simple.

- *mongoose*[11]: Es una herramienta de modelado de objetos de *MongoDB*, que proporciona conversión de tipos, validación y creación de consultas. Es la biblioteca utilizada para acceder a la base de datos desde el servidor web.

La estructura de los ficheros de la aplicación es como sigue (véase la Figura 4.16):

[10] https://expressjs.com/
[11] https://mongoosejs.com/

- **models:** Carpeta que contiene todos los modelos de *mongoose*. Un modelo es un constructor que toma un esquema y crea una instancia de un documento de *MongoDB*. En esta carpeta, hay un modelo por cada colección de la base de datos. Cada modelo utiliza un esquema de la colección que representa, donde un esquema representa la estructura de los datos del documento, que se aplica a través de la capa de aplicación.

- **server.js** Archivo que contiene el programa principal, el cual se encuentra a la escucha de las solicitudes del usuario web. Este programa principal se conecta a la base de datos utilizando *mongoose* y después se mantiene a la espera de solicitudes del cliente web. Por cada solicitud del cliente, el servidor utilizando *Expressjs* enruta la solicitud a uno de los puntos finales definidos en el archivo *api.js*, el cual resuelve la solicitud.

- **api.js** Implementa una serie de métodos para resolver las solicitudes recibidas. Los métodos son de tipo *get()* para manejar solicitudes *GET* y *post()* para manejar solicitudes *POST* de *HTTP*. Cada método, según la lógica que tenga implementada, realiza una o varias consultas a la base datos utilizando los modelos definidos en la carpeta *models*. Para la respuesta a las solicitudes utiliza la *API REST* por medio de *HTTP* en formato *JSON*.

4.1.4.2. Cliente web

Esta aplicación implementa la interfaz del sistema a la que acceden los usuarios mediante un navegador web. La aplicación se ejecuta en un *pod* de *Kubernetes*, exponiendo un puerto (externo al servidor) para que los usuarios con la *IP* del servidor y este puerto puedan acceder a la interfaz.

El cliente web fue programado con *Reactjs* y utiliza los siguientes bibliotecas:

- *Material UI*[12]: *Framework* de diseño de interfaces de usuario, basado en componentes para *reactJS* y estilos *material desing Google*.

- axios[13]: Biblioteca *JavaScript* que ejecuta un cliente *HTTP* ligero basado en la API *Promise* nativa de *Javascript* [36], lo que le permite aprovechar *async* y *await* de *JavaScript*. Además, permite configurar y realizar solicitudes a un servidor, recibiendo respuestas fáciles de procesar.

La interfaz web se actualiza cada 10 segundos utilizando el código mostrado en la Figura 4.17. En este código, el método *componentDidMount()* se llama durante la fase de montaje del ciclo de vida de *React*, lo que permite cada 10 segundos *renderizar* el componente, llamando al método *get()*, que solicita al servidor todos los datos actualizados que necesita la interfaz.

A continuación, se proporcionan detalles específicos sobre la autenticación y el control de acceso al sistema, así como acerca del diseño de la interfaz de usuario.

Autenticación y control de acceso

[12] https://material-ui.com/
[13] https://axios-http.com/

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.get(),
    10000
  );
}
```

Figura 4.17: Procedimiento *componentDidMount()* del cliente web que permite actualizar la interfaz web periódicamente.

Los usuarios del sistema se autenticarán mediante un nombre de cuenta y una contraseña para garantizar una cierta seguridad. El sistema no cuenta con un apartado de inserción y edición de usuarios, este aspecto se tratará en versiones futuras. Así, en el desarrollo actual, para insertar un nuevo usuario con un rol específico, el administrador lo insertará directamente en la base de datos.

Los roles condicionarán las funcionalidades del portal a las que tendrá acceso cada usuario. Los roles definidos en el portal son los siguientes:

- **Administrador:** Tiene acceso al apartado de Espacios y Sensores donde podrán insertar o eliminar las ubicaciones y los sensores desplegados, respectivamente.
- **Estándar:** Tiene acceso al apartado de Sensores e Inicio donde podrá visualizar el comportamiento y estadísticas de cada sensor, pero no podrá editar ninguna configuración.

Interfaz de usuario

El acceso al *frontend* por el usuario se realiza a través de una página de *login*, en la que el usuario introduce un usuario y una contraseña válida. Una vez autenticado, el usuario accede al portal que le corresponda según su rol. Véase la Figura 4.18. A continuación, se expondrán las partes de la interfaz de usuario a la que tiene acceso cada usuario acorde a su rol:

Figura 4.18: Vista interfaz de autenticación de usuario.

- **Usuario administrador:** Tal y como se muestra en el mapa de rutas en la Figura 4.19, este usuario tiene acceso a:

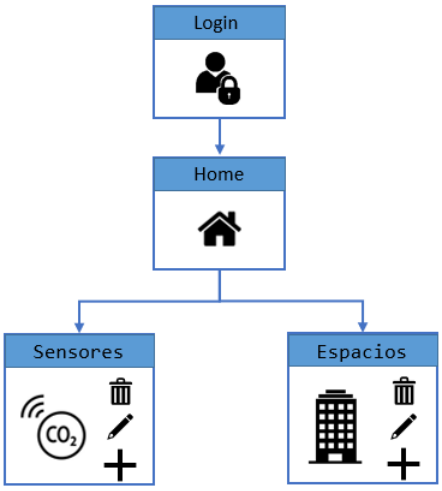

Figura 4.19: Mapa de rutas a las que tiene acceso en la interfaz un usuario con rol de administrador.

- **Sensores:** Permite gestionar los aspectos relativos a la configuración de los elementos sensores. Véase la Figura 4.20. En esta interfaz, el administrador podrá visualizar con una tabla los sensores presentes en el sistema, pudiendo insertarlos o eliminarlos, además de poner en marcha el sensor para que el controlador ejecute un escuchador de cara a recolectar datos y almacenarlos en la base de datos.

Figura 4.20: Apartado Sensor

La interfaz haciendo uso de *axios* envía al servidor web las peticiones *GET* o *POST*, para tener acceso a las colecciones Sensor y Espacios de la base de datos. Un ejemplo de una petición al servidor se encuentra en la Figura 4.21, donde se solicita la lista de todos los sensores del sistema.

Las acciones que pueden llevar a cabo los administradores en este punto de la interfaz son las siguientes:

```
getSensores = () => {
  axios.get("/getAllSensores")
    .then((response) => {
      const data = response.data;
      this.setState({
        sensores: data
      });
    })
    .catch(() => {
    });
};
```

Figura 4.21: Ejemplo de una petición desde el cliente web al servidor.

- Insertar un nuevo Sensor.
- Eliminar un sensor.
- Ejecutar un sensor.
- Ordenar la tabla de sensores por *id*, espacio o estado del sensor.

En la ventana de añadir un nuevo sensor, siempre que exista en el sistema al menos un espacio, el administrador tiene que proporcionar los siguientes datos mediante un formulario:

- Seleccionar de una lista desplegable el nivel del Espacio donde quiere colocar el sensor.
- Seleccionar de una lista desplegable el nombre del Espacio donde quiere colocar el sensor.

El resto de datos del documento por insertar en la colección Sensor se autocompletan por la aplicación, generando un identificador único, configurando el estado como *escuchando*, asociación del espacio elegido con su identificador. Además, cuando se añade un sensor, se actualiza el campo *estado* del documento correspondiente de la colección *Espacios*, tomando un valor 1, de modo que tal espacio no pueda ser eliminado hasta que no se eliminen todos los sensores asignados. Como se espera, al eliminar el último sensor asociado a un espacio, el campo *estado* del documento correspondiente tomará un valor de 0.

- **Espacios:** Permite gestionar los aspectos relativos a la configuración de los espacios (lugar donde se despliega un sensor). Véase la Figura 4.22. En este punto de la interfaz, el administrador podrá visualizar con una tabla los espacios presentes en el sistema, pudiendo insertar o eliminar cada uno de ellos.

  Los espacios, como se vio durante el diseño de la estructura de datos de la base de datos, se guardan como documentos, pero en la interfaz se visualizan en forma de árbol, ya que los espacios se dividen por niveles, donde cada nivel tiene uno o varios niveles superiores. El objetivo de separar en niveles es que la concatenación del nivel principal hasta el final sea usado por el sensor, para saber en qué *topic* del *broker* publicar sus datos.

  Las acciones que pueden llevar a cabo los administradores en este punto de la interfaz son las siguientes:

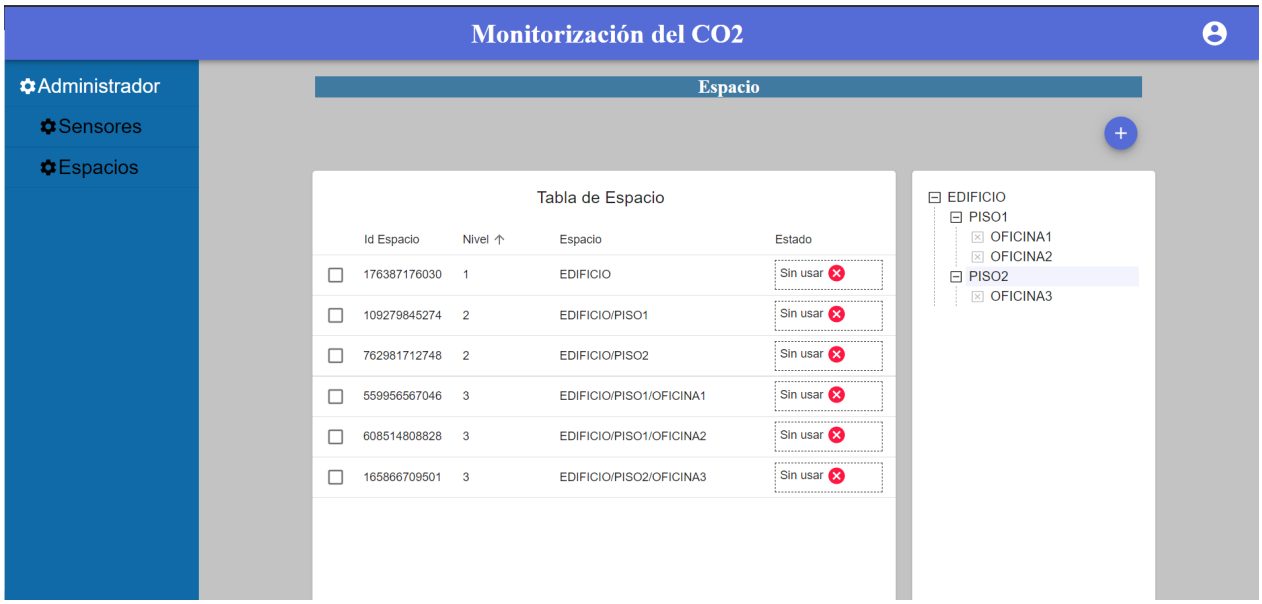

Figura 4.22: Apartado Espacio

- Insertar un nuevo Espacio.
- Eliminar un Espacio.
- Ordenar la tabla de sensores por *id*, nivel, espacio o estado del espacio.
- Visualizar la estructura de los espacios en forma de árbol.

En la ventana de añadir un nuevo espacio, el administrador tiene que proporcionar los siguientes datos mediante un formulario:

- Seleccionar de una lista desplegable el nivel superior del nuevo espacio que se quiere añadir.
- Seleccionar de una lista desplegable el nombre del Espacio superior del nuevo espacio que se quiere añadir.
- Seleccionar de una lista desplegable el número del nuevo nivel, éste se autogenera como el siguiente nivel del seleccionado como superior.
- Añadir el nombre del nivel actual donde el campo es de tipo *string(25)*, permitiendo solo letras y números sin espacios. El nombre se convierte a mayúscula automáticamente para evitar errores con los nombres, ya que dentro de un mismo nivel no se puede repetir el nombre.

El resto de datos del documento se autocompleta por el sistema, generando un identificador único, guardando el identificador, el nivel añadido, el nombre añadido y el identificador del nivel superior.

- **Usuario estándar:** Tal y como se muestra en el mapa de rutas en la Figura 4.23, este usuario tiene acceso a la visualización de datos capturados por los sensores y las ubicaciones disponibles. En todo momento, el usuario tendrá disponible una visualización de posibles alertas, en caso de que alguno de los sensores supere los niveles permitidos de CO_2, tal y como se observa en la Figura 4.24. Las pantallas *Inicio* y *Espacios* son como siguen:

  - **Inicio:** En esta pantalla los usuarios pueden visualizar un gráfico con la media de CO_2 por hora del espacio y fecha filtrada, mostrando la media por cada

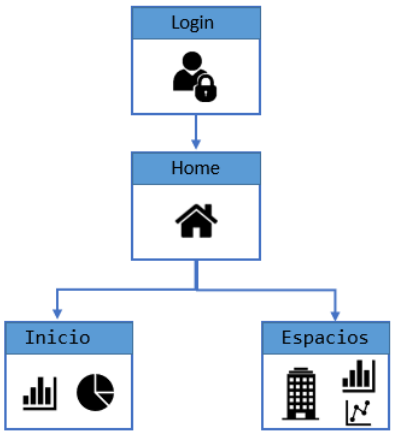

Figura 4.23: Mapa de rutas a las que tiene acceso en la interfaz un usuario estándar.

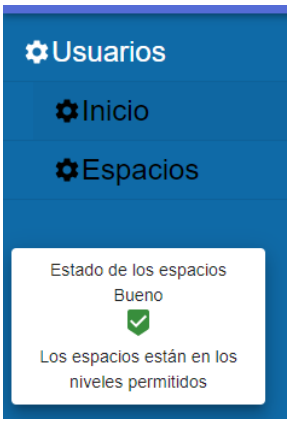

Figura 4.24: Visualización de posibles alertas en el menú de usuarios estándar.

sensor asignado al espacio. Además pueden visualizar la media general del CO_2 en ppm de todo el sistema.

- **Espacios:** Esta pantalla permite visualizar todos los sensores activos del sistema. Véase la Figura 4.25. Esta información se divide en tres tablas:
  - Tabla de espacios: Recoge todos los sensores activos del sistema, mostrando el nombre del espacio, *id* del sensor, nivel de CO_2 y la ultima actualización disponible. Esta tabla permite filtrar por cada uno de los campos mostrados.
  - Tabla de espacios que superan los niveles de CO_2 permitidos: Recoge todos los espacios que superan los niveles de CO_2 permitidos, mostrando el nombre del espacio y el nivel de CO_2 en ppm.
  - Espacio con nivel de CO_2 mas alto: Muestra el espacio con mayor nivel de CO_2 de todos los del sistema. Para ello utiliza el código mostrado en la Figura 4.26, donde se envía una solicitud al servidor utilizando *axios*.

En la interfaz del usuario Estándar es donde se aplica la lógica que evalúa si los niveles de CO_2 entran en una de las tres categorías de evaluación (bueno, medio o malo) como se vio en la Sección 2.2.

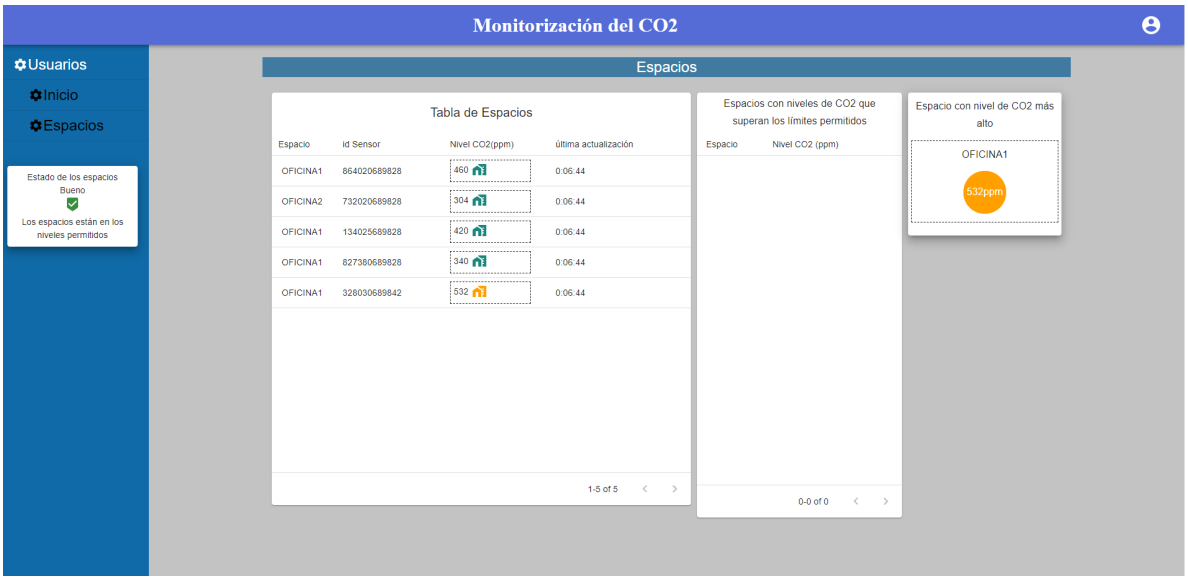

Figura 4.25: Pantalla de espacios en el usuario con rol estándar.

```
getMasAlto = () => {
  axios.get("/getDataMasAlto")
      .then((response) => {
          const data = response.data;
          this.setState({
              masAlto: data[0].masV,
              nameMasAlto: data[0].masN
          });
      })
      .catch(() => {
      });
};
```

Figura 4.26: Código que permite mostrar el espacio con mayor nivel de CO_2 de todos los del sistema.

4.2. Implementación del sistema sensor

Como se describió previamente, el sistema sensor está compuesto por dos componentes (véase la Figura 4.27) como siguen:

- **NodeMCU:** Es el nodo de procesamiento IoT implementado sobre la placa de desarrollo *NodeMCU ESP8266MOD-12F*, que recolectará la información acerca del CO_2 para después enviarla al *Broker* MQTT vía WiFi.

- **MQ-135:** Es el sensor encargado de capturar la variación de CO_2 en el entorno. Esta información será recolectada por el *NodeMCU* mediante una conexión física basada en nivel de tensión.

En las siguientes secciones se describen la conexión física entre ambos componentes, además de la descripción del programa principal del NodeMCU y finalmente la comunicación entre el sistema de sensores y el sistema de servidor.

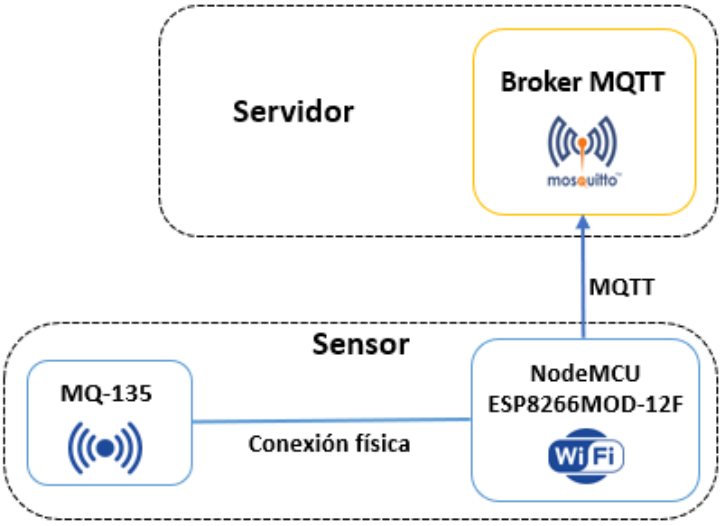

Figura 4.27: Interacción del sistema sensor con los componentes que lo conforman y con el sistema servidor.

4.2.1. Conexión física de los componentes

Tal y como se describió en la Sección 2.3.3, el sensor *MQ-135* consta de cuatro pines (*Vcc*, *ground*, una salida digital y una salida analógica), optando en este proyecto por utilizar la salida analógica, por lo que se requiere el uso del periférico ADC del NodeMCU siguiendo la distribución de pines de ambos componentes (véase el *pinout* del NodeMCU descrito en la Sección 2.3.2). Por tanto, la conexión física entre ambos componentes se lleva a cabo como sigue (véase su representación esquemática en la Figura 4.28):

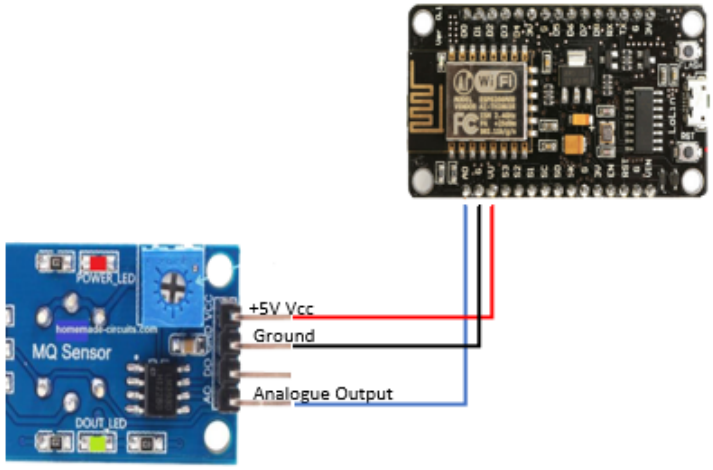

Figura 4.28: Representación esquemática de la conexión física entre sensor MQ-135 y el NodeMCU ESP8266MOD-12F

- El pin 1 (*Vcc*) del sensor se conecta físicamente con al pin *VU* del NodeMCU, para proporcionar una tensión de alimentación al sensor de 5V, que es la requerida por éste según las especificaciones técnicas.
- El pin 2 (*ground*) del sensor se conecta físicamente con el pin *GND* del *NodeMCU*, de modo que el sensor tenga el mismo punto de referencia de voltaje que el *NodeMCU*.

- El pin 4 (salida analógica) del sensor se conecta físicamente con el pin *A0* del *NodeMCU*, que es el puerto de entrada al periférico *ADC* implementado dentro del NodeMCU. Este periférico convertirá el valor de tensión (en el rango 0-5v) proporcionado por el sensor en niveles digitales que irán desde el 0 al 1023, acorde a la resolución del ADC (8bits).

La conexión de los dos componentes que conforman el sistema sensor se inspiró en el artículo [37]. El aspecto resultante de la conexión de ambos componentes se puede observar en la Figura 4.29.

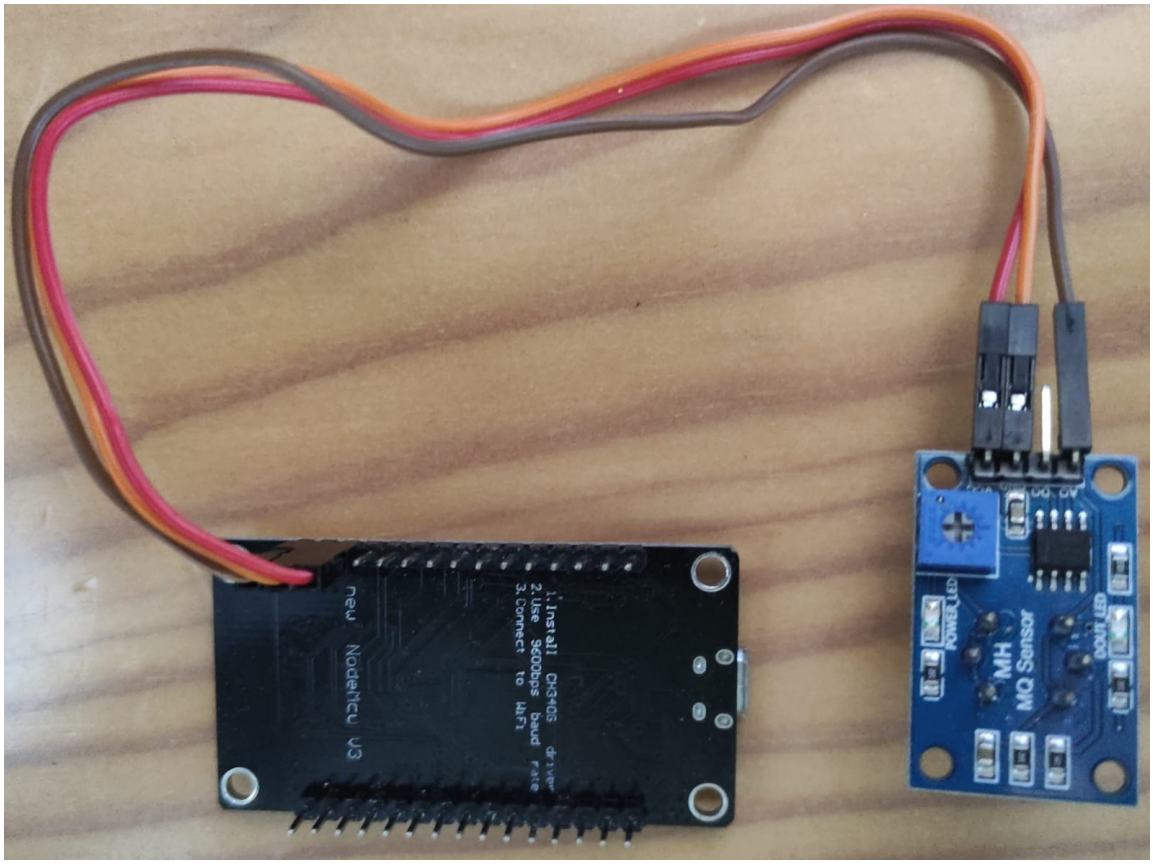

Figura 4.29: Conexión física entre sensor MQ-135 y el NodeMCU ESP8266MOD-12F

4.2.2. Programa principal del NodeMCU

El programa principal o *firmware* del nodo de procesamiento *NodeMCU* se ha llevado a cabo con la plataforma *PlatformIO*, utilizando la combinación de los lenguajes de programación C y C++. La estructura de ficheros de este desarrollo se muestra en la Figura 4.30 y se detalla a continuación:

- **platformio.ini:** Archivo de configuración del proyecto en *PlatformIO*, donde se definen las opciones de dependencias, opciones de compilación, opciones de carga y otras opciones avanzadas.
- **MQ135.cpp y MQ135.h:** Biblioteca de *Arduino* para el sensor MQ-135 [14]. Esta

[14] https://github.com/GeorgK/MQ135

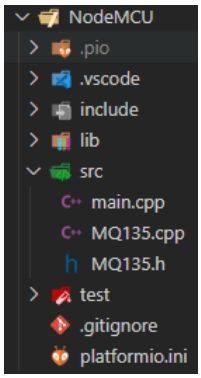

Figura 4.30: Estructura de ficheros en el desarrollo del programa principal del NodeMCU.

biblioteca se encarga de hacer todos los cálculos necesarios para primero calibrar el sensor y después hacer una lectura correcta sobre éste.

- **main.cpp:** Fichero principal. Contiene toda la lógica que ejecuta el *NodeMCU*. El contenido del fichero se explica a continuación.

El fichero **main.cpp:** incluye los siguientes métodos desarrollados:

- **setup_wifi()** Función que configura la conexión WiFi del *NodeMCU* con el propósito de poder enviar información al *Broker* MQTT del sistema servidor. Este método se verá en detalle en la siguiente sección.
- **reconnect()** Método por el cual, en caso de perdida de la conexión con el *Broker* MQTT, se hace un reintento automático. En caso de no conseguir restablecer la conexión, se queda en bucle infinito. Para este caso, el control de caídas es llevado por el módulo Controlador que se explicara más adelante.
- **setup()** Función que se llama cuando comienza la ejecución del *NodeMCU*. Aquí se inicializan las variables necesarias en el momento del arranque. En esta función se inicializa la variable *configuracionInicial* a *true*, además también se establece la conexión con la red WiFi y la conexión con el cliente MQTT que se verá más adelante.
- **obtenerRZERO()** Función que obtiene el *RZero*. El *RZero* es el valor de la resistencia para la calibración del sensor. Este valor se obtiene utilizando la función *getRZero()* de la biblioteca de *Arduino* para el sensor MQ-135. Esta función, que se muestra en la Figura 4.31 tiene el siguiente funcionamiento:

```
float MQ135::getRZero() {
  return getResistance() * pow((ATMOCO2/A_VALOR), (1./B_VALOR));
}
```

Figura 4.31: Función *getRZero()* de la biblioteca de *Arduino* para el sensor MQ-135.

1. Mediante la función *getResistance()* en la Figura 4.32, se calcula el valor de la resistencia acorde a la presencial del gas detectado según la expresión dada por

```
float MQ135::getResistance() {
  int val = analogRead(_pin);
  return ((1023./(float)val) - 1.)*RL;
}
```

Figura 4.32: Función *getResistance()* de la biblioteca de *Arduino* para el sensor MQ-135.

$$R = \left(\left(\frac{1023}{val}\right) - 1\right) \cdot RL, \tag{4.1}$$

donde val es el nivel leído del ADC, RL es la resistencia de carga en la placa, tomando un valor de 1k Ohm (por lo que RL=1000), y 1023 es el nivel máximo del ADC[15].

2. A continuación, se multiplica el valor obtenido para la resistencia en el paso previo (R) para obtener el *RZero* según la expresión dada por

$$RZero = R \cdot \left(\left(\frac{ATMOCO2}{A\_VALOR}\right)\left(\frac{1}{B\_VALOR}\right)\right), \tag{4.2}$$

donde ATMOCO2 es el nivel de CO_2 atmosférico usado como referencia[16] (ATMOCO2 = 410) y A_VALOR= 116.6020682 y B_VALOR= 2.769034857 son parámetros definidos por la la biblioteca acorde a la curva de sensibilidad del sensor (véase la Figura 2.9 [38]).

- **setMediaRzero()** Función que calcula la media del valor *RZero*. Para ello, cada 10 segundos y durante 6 iteraciones, se calcula la media del *RZero* utilizando la función *obtenerRZERO*. El código de la función *setMediaRzero()* se encuentra en la Figura 4.33.

- **obtenerPPM()** Función que obtiene el valor de concentración de CO_2 en ppm. Para ello utiliza la función *getPPM()* de la biblioteca, la cual se muestra en la Figura 4.34. Esta función ejecuta la expresión dada según

$$PPM = A\_VALOR * \left(\frac{getResistance()}{rzero}\right)^{-B\_VALOR}, \tag{4.3}$$

donde *rzero* es la media de la resistencia calculada por *setMediaRzero()* y A_VALOR y B_VALOR toman los valores definidos previamente.

- **loop()** Función principal, que implementa un bucle infinito según el diagrama de flujo en la Figura 4.35. En éste se observa como primero, mediante la función **setup()** se inicializan las variables necesarias en el momento del arranque para después pasar a ejecutarse la función **loop()**. Aquí, primero verifica si está en la configuración inicial (configuración que se ejecuta una vez) y si lo está, ejecuta la función **setMediaRzero()** vista anteriormente. En el caso de no estarlo, se comprueba si el cliente MQTT está conectado, si no lo esta, se ejecuta la función **reconnect()** vista anteriormente.

[15] https://www.arduino.cc/reference/en/language/functions/analog-io/analogread/
[16] https://www.co2.earth/

```
void setMediaRzero()
{
  float _samaRzero = 0;
  int _cont = 6;
  for (int i = 0; i < _cont; i++)
  {
    _samaRzero = _samaRzero + obtenerRZERO();
    delay(10000);
  }
  float _mediaRzero = _samaRzero / _cont;

  Serial.print("--> Valor de la Media de RZero: ");
  Serial.println(_mediaRzero);
  gasSensor.setRZero(_mediaRzero);
}
```

Figura 4.33: Función *setMediaRzero()* de la biblioteca de *Arduino* para el sensor MQ-135.

```
float MQ135::getPPM() {
  return A_VALOR * pow((getResistance()/rzero), -B_VALOR);
}
```

Figura 4.34: Función *getPPM()* de la biblioteca de *Arduino* para el sensor MQ-135.

Si el cliente está conectado ejecuta la funcionalidad principal de la función, es decir, obtener la concentración de CO_2 en ppm mediante la función **obtenerPPM()** para después, enviarlo al servidor en el *topic* especificado, el siguiente código lo muestra:

```
client.publish(topic, ppm)
```

Por último, la función espera 30 segundos para después volver a ejecutarse.

4.2.3. Comunicación del sistema sensor con el sistema servidor

El NodeMCU consta de un módulo WIFI que permite la conexión del sistema sensor a internet. Utilizando esta característica, NodeMCU podrá enviar los datos recogidos por el MQ-135 al Broker MQTT del sistema servidor. Para ello, el sistema sensor tendrá que conocer la dirección *IP* del servidor, el nombre de la red WiFi y su contraseña, así como el puerto expuesto al exterior para el *pod* de *Kubernetes* que ejecuta el *broker Mosquitto*, tal y como se muestra en la Figura 4.36.

La conexión del *NodeMCU* a la red WiFi se realiza mediante la biblioteca *ESP8266WiFi*[17]. El NodeMCU ejecuta la función *setup_wifi()*, descrita previamente, cuyo contenido puede observarse en la Figura 4.37. En caso de perdida de la conexión, se utilizará la función *reconnect()*.

[17] https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266WiFi

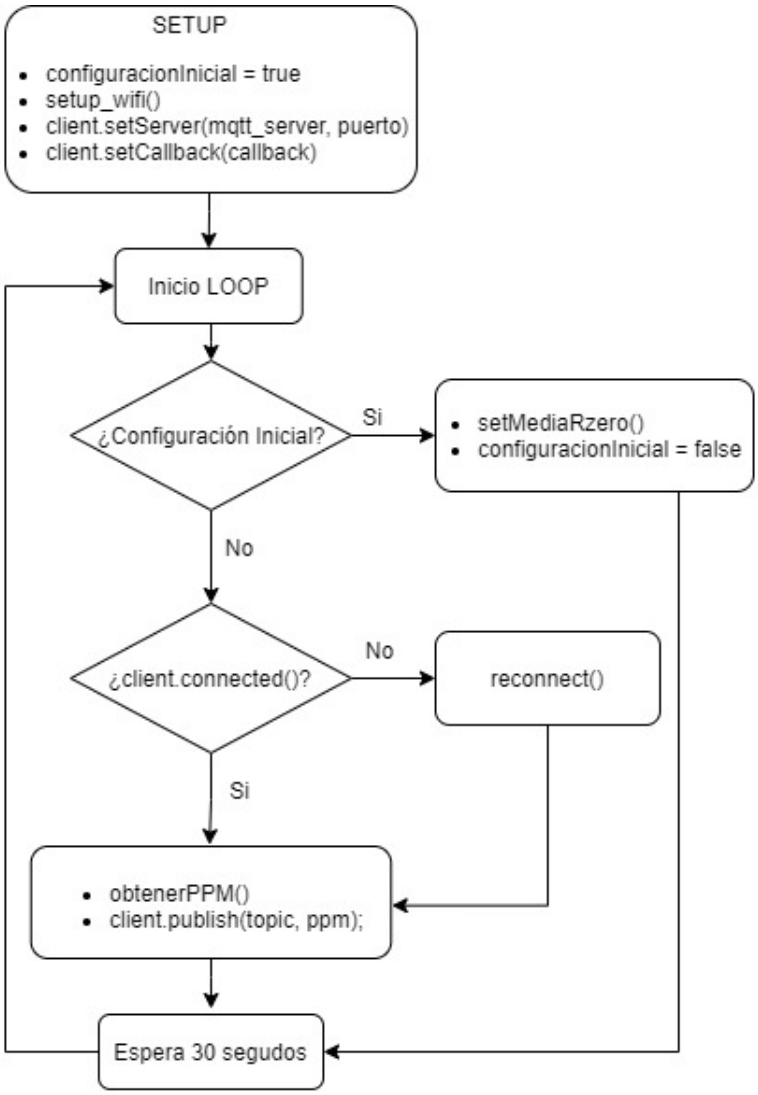

Figura 4.35: Diagrama de flujo de la función *loop()* del NodeMCU.

Una vez que el *NodeMCU* se encuentra conectado a la red WiFi, en la función *setup()* descrita previamente se inicia una conexión con el *Broker* MQTT utilizando la biblioteca *PubSubClient* [18]. La biblioteca *PubSubClient* importada en el fichero de configuración *platformio.ini* permite enviar y recibir mensajes MQTT. Para inicial la conexión con el *broker*, primero se define la variable global **cliente** del tipo *PubSubClient* tal y como se observa en la Figura 4.37. Después, en la función *setup()* se ejecuta el siguiente código:

```
client.setServer(mqtt_server, 30210)
```

el cual añade al cliente MQTT la dirección IP del servidor y el puerto del *broker* MQTT. En la función *loop()*, tal y como se ve en código siguiente:

```
ppm = obtenerPPM();
String jsonDato = "{\"id\":\"" + ID + "\", \"dato\":\"" + ppm + "\"}";
client.publish(MQTT_TOPIC, String(jsonDato).c_str());
```

[18] https://platformio.org/lib/show/89/PubSubClient

```
const char *ssid = "nombre del las red wifi";
const char *password = "contraseña";
const char *mqtt_server = "dirección ip del servidor";
int puerto = "puerto expuesto";
```

Figura 4.36: Código de configuración de la conexión mediante WIFI del sistema sensor al sistema servidor.

```
WiFiClient espClient;
PubSubClient client(espClient);

void setup_wifi()
{
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);
}
```

Figura 4.37: Código de la función *setup_wifi()*.

se obtiene el dato recogido por el sensor, usando la función *obtenerPPM()* y se envía la información al *Broker* MQTT usando la función *publish()* de la biblioteca *PubSubClient*. Para ello, previamente los datos se encapsulan en un JSON con la estructura siguiente:

```
{
    "id": ID
    "dato": ppm
}
```

donde el *ID* es el identificador del *NodeMCU*, autogenerado por el sistema en el modulo *Web Frontend* durante la inserción del sensor en el sistema, y dato es la medición de concentración obtenida por el sensor.

Capítulo 5

Conclusiones

El uso de sensores en el ámbito de la creación de sistemas enfocados en el IoT para la monitorización de magnitudes físicas o químicas puede aportar interesantes ventajas. Por ejemplo, en este proyecto se ha permitido aportar una solución para la monitorización de los niveles de CO_2 en espacios cerrados de forma eficiente, rápida, económica y escalable.

El uso de kubernetes para el despliegue de todos los componentes del sistema servidor ha permitido aislar fácilmente los procesos del sistema, además de facilitar el rápido despliegue del sistema sin importar el proveedor de servicios de *Cloud* o servidor local. Este aspecto sin duda es de gran importancia para que la solución sea escalable según las necesidades futuras, pudiendo migrar el sistema servidor de forma sencilla.

Con la combinación del sensor MQ-135 y el NodeMCU, se ha obtenido una solución para la monitorización del CO_2 en espacios cerrados de bajo coste y de rápido despliegue. Sin embargo, el bajo coste del sensor implica limitaciones en la propia capacidad de detección del sensor, ya que no solo detecta CO_2. Esta limitación sería fácilmente abordable mediante el uso de un nuevo sensor en combinación con el NodeMCU gracias a que el sistema sensor está compuesto por dos componentes independientes. Por otro lado, el diseño de una arquitectura donde el sistema sensor y el sistema servidor se comunican vía Internet sobre el protocolo MQTT, permite que el sistema sensor sea perfectamente intercambiable por otro de mayor capacidad computacional si se requiere.

En cuanto a los objetivos propuestos en el proyecto, se puede afirmar que se ha cumplido el objetivo principal sobre la de creación de este sistema de monitorización, creando una solución económica que ayuda a la cuantificación de los niveles del CO_2 en espacios cerrados no industrializados, permitiendo administrar los datos recolectados por múltiples sistemas sensores.

Es relevante reseñar la relevancia del sistema para evitar contagios por el coronavirus SARS-CoV-2 en espacios cerrados. Con este sistema, los usuarios podrían identificar los espacios que superan los niveles de CO_2 permitidos, para así poder tomar medidas como la ventilación del área, ya sea por medios naturales o mecánicos. También se debe reseñar que este sistema tiene utilidad una vez se supere la alerta generada por la pandemia, ya que la monitorización de la calidad del aire en espacios cerrados es un aspecto vital para la salud humana.

En el plano personal, gracias a este proyecto he adquirido conocimientos del desarrollo de soluciones enfocadas en el IoT, especialmente en la sensorización de espacios cerrados. Cada nueva funcionalidad implementada en el sistema ha conllevado algún nuevo aprendizaje que complementa la formación obtenida durante la titulación.

Capítulo 6

Líneas futuras

El proyecto desarrollado es susceptible de ser mejorado en distintos aspectos, aunque cumple con la idea principal. A continuación, se describen algunas de las ideas para futuros desarrollos:

- El primer aspecto a mejorar es el escalado de los sensores, ya que estos solo se conectan y envían datos dentro de una *intranet*. La solución seria implementar una puerta de enlace MQTT o MQTT *Gateway*, que permita la conexión de los sensores a internet y de esta forma conectarse al servidor.
- En segundo lugar, es importante garantizar la seguridad del protocolo MQTT, que es utilizado por los sensores para comunicarse con el servidor. Actualmente, los mensajes MQTT se envían sin ningún tipo de cifrado. La solución puede pasar por las siguientes opciones:
  - **A nivel de red:** Proveer una conexión fiable, usando una red física segura o VPN.
  - **A nivel de transporte:** Implementar encriptación SSL/TLS, para la encriptación del transporte de los datos.
  - **A nivel de aplicación:** Utilizar credenciales de nombre de usuario/contraseña o la encriptación de la carga útil.

  Además, si se implementa una puerta de enlace MQTT ,se puede utilizar un método de comunicación más seguro entre éste y el servidor.
- En tercer lugar, dado que en el *Front-end* se implementa la API REST a través de HTTP en lugar de HTTPs, las solicitudes se envían a través de texto plano, dando lugar a un problema de seguridad. Por lo antes expuesto es importante implementar el protocolo HTTPS en el servidor para proporcionar seguridad en la transferencia de los datos. Además, otra vulnerabilidad de la API REST es que no se han hecho uso de tokens para verificar la autenticidad de los usuarios en cualquier momento. Para solucionarlo, se podría crear una componente para verificar los tokens en las peticiones REST a la API.
- En cuarto lugar, como los usuarios del sistema son insertados por en usuario administrador directamente a la base de datos, sería interesante implementar en versiones futuras del *Front-end* un apartado para añadir, editar o eliminar los usuarios del sistema.

- Por último, la interfaz gráfica del *Front-end* se podría enriquecer para hacerla más atractiva para los usuarios.

Bibliografía

[1] Protect Soiart Distribución. Medidor de co2 dioxcare pdf. https://protect.soiartdistribucion.com/producto/medidor-de-co%E2%82%82-dioxcare/.

[2] Medidores de co2 de alta precisión con conexión a internet control de calidad del aire. https://kipmion.com/wp-content/uploads/2021/05/Catalogo-CO2Panel-PI.pdf.

[3] AZ-Delivery. Módulo nodemcu lua lolin v3 con esp8266 12f. https://www.az-delivery.de/es/products/nodemcu-lolin-v3-kostenfreies-e-book.

[4] AZ-Delivery. Mq-135 gas sensor modul. https://www.az-delivery.de/products/mq-135-kostenfreies-e-book.

[5] Technical data mq-135 gas sensor. https://www.olimex.com/Products/Components/Sensors/Gas/SNS-MQ135/resources/SNS-MQ135.pdf.

[6] Departamento de Salud Ambiental Subdirección General de Salud Pública (2020). Medición de la concentración de co2 como indicador de una ventilación adecuada de edificios y locales. covid19. https://madridsalud.es/wp-content/uploads/2020/11/InfSAM33-2020Ventilacion_interio_como_medida_preventivaCOVID19.pdf. Accessed: 2020-10-01.

[7] Syahrun Neizam Mohd Dzullkiflli, Lee Yee Yong, Abdul Mutalib Leman, Samiullah Sohu, et al. A study of indoor air quality in refurbished museum building. *Civil Engineering Journal*, 4(11):2596–2605, 2018.

[8] Leyla Morán, Guisela Yábar, and Krupuskaya Figueroa. Calidad del aire interior en el síndrome del edificio enfermo, ciudad de trujillo. *Revista de la Facultad de Medicina Humana*, 17(4), 2017.

[9] Dongwan Kim, Joo-Yeon Lee, Jeong-Sun Yang, Jun Won Kim, V Narry Kim, and Hyeshik Chang. The architecture of sars-cov-2 transcriptome. *Cell*, 181(4):914–921, 2020.

[10] Antonio Alcamí, Margarita del Val, Miguel Hernán, Pello Latassa, José Luis Jiménez, Xavier Querol, Ana Robustillo, Gloria Sánchez, and Alfonso Valencia. Informe científico sobre vías de transmisión sars-cov-2. 2020.

[11] María Cruz Minguillón, Xavier Querol, José Manuel Felisi, and Tomás Garrido. Guía para ventilación de las aulas csic. 2020.

[12] Wan Haslina Hassan et al. Current research on internet of things (iot) security: A survey. *Computer networks*, 148:283–294, 2019.

[13] Andrés Pérez-Estaún, Manuel Gómez, and Jesús Carrera. El almacenamiento geológico de co2, una de las soluciones al efecto invernadero. *Enseñanza de las Ciencias de la Tierra*, 17(2):179–189, 2009.

[14] Sean O'Neill. Global co2 emissions level off in 2019, with a drop predicted in 2020. *Engineering (Beijing, China)*, 2020.

[15] José Humberto Mondragón-Suárez, Alfredo Sandoval-Villalbazo, and Fernanda Breña-Ramos. Calentamiento global: una secuencia didáctica. *Revista Mexicana de Física E*, 65(1 Jan-Jun):52–57, 2019.

[16] Edgar Enrique La Rotta Villamizar and Jarol Derley Ramón Valencia. Análisis de daños estructurales en edificaciones por contaminación del dióxido de carbono (co2) asociado al flujo vehicular en la vía nacional en el casco urbano del municipio de pamplona, norte de santander. *BISTUA REVISTA DE LA FACULTAD DE CIENCIAS BASICAS*, 16(2):145–152, 2018.

[17] Comercio y Turismo y el Ministerio de Sanidad Ministerio de Industria. Recomendaciones de operación y mantenimiento de los sistemas de climatización y ventilación de edificios y locales para la prevención de la propagación del sars-cov-2. https://www.mscbs.gob.es/profesionales/saludPublica/ccayes/alertasActual/nCov/documentos/Recomendaciones_de_operacion_y_mantenimiento.pdf. 2020-06-30.

[18] Edwing A Almeida Calderón and Marcela E Buitrón de la Torre. El internet de las cosas y el diseño del futuro the internet of things and the future design.

[19] Zhi-Hong Qian and Yi-jun Wang. Iot technology and application. *Acta Electronica Sinica*, 40(5):1023–1029, 2012.

[20] Dipa Soni and Ashwin Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*, volume 20, 2017.

[21] Gurkan Tuna, Dimitrios G Kogias, V Cagri Gungor, Cengiz Gezer, Erhan Taşkın, and Erman Ayday. A survey on information security threats and solutions for machine to machine (m2m) communications. *Journal of Parallel and Distributed Computing*, 109:142–154, 2017.

[22] Juhasova Bohuslava, Juhas Martin, and Halenar Igor. Tcp/ip protocol utilisation in process of dynamic control of robotic cell according industry 4.0 concept. In *2017 IEEE 15th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 000217–000222. IEEE, 2017.

[23] Mario E Cueva Hurtado and Diego Javier Alvarado Sarango. Análisis de certificados ssl/tls gratuitos y su implementación como mecanismo de seguridad en servidores de aplicación. *Enfoque UTE*, 8:273–286, 2017.

[24] Roger A Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13):265, 2017.

[25] Monika Kashyap, Vidushi Sharma, and Neeti Gupta. Taking mqtt and nodemcu to iot: communication in internet of things. *Procedia computer science*, 132:1611–1618, 2018.

[26] Ebenezer Hailemariam, Rhys Goldstein, Ramtin Attar, and Azam Khan. Real-time occupancy detection using decision trees with multiple sensor types. In *Proceedings of the 2011 Symposium on Simulation for Architecture and Urban Design*, pages 141–148, 2011.

[27] Kinnera Bharath Kumar Sai, Subhaditya Mukherjee, and H Parveen Sultana. Low cost iot based air quality monitoring setup using arduino and mq series sensors with dataset analysis. *Procedia Computer Science*, 165:322–327, 2019.

[28] Sergio Noriega. Familias lógicas. *Presentado en Introducción a los Sistemas Lógicos y Digitales*, 2008.

[29] Mahmood Moussavi, Yasaman Amannejad, Mohammad Moshirpour, Emily Marasco, and Laleh Behjat. Importance of data analytics for improving teaching and learning methods. In *Data management and analysis*, pages 91–101. Springer, 2020.

[30] José Rafael Capacho and Wilson Nieto Bernal. *Diseño de bases de datos*. Universidad del Norte, 2017.

[31] Shannon Bradshaw, Eoin Brazil, and Kristina Chodorow. *Mongodb: the definitive guide: powerful and scalable data storage*. O'Reilly Media, 2019.

[32] Tom Marrs. *JSON at work: practical data integration for the web*. .º'Reilly Media, Inc.", 2017.

[33] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[34] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: up and running: dive into the future of infrastructure*. O'Reilly Media, 2019.

[35] Shyue Ping Ong, Shreyas Cholia, Anubhav Jain, Miriam Brafman, Dan Gunter, Gerbrand Ceder, and Kristin A Persson. The materials application programming interface (api): A simple, flexible and efficient api for materials data based on representational state transfer (rest) principles. *Computational Materials Science*, 97:209–215, 2015.

[36] Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. Refactoring asynchrony in javascript. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 353–363. IEEE, 2017.

[37] Bakti Dwi Waluyo, Harvei Desmon Hutahaean, and Agus Junaidi. Multiplexer performance testing for iot-based air quality monitoring system: Multiplexer performance testing for iot-based air quality monitoring system. *Jurnal Mantik*, 4(1):33–40, 2020.

[38] Y LAKSHMI NARAYANA REDDY and DR TJ NAGA LAKSHMI. Iot based air quality monitoring system using mq135 and arduino.

Apéndice A

Anexo I: Presupuesto

En este apartado, se especifica el presupuesto para el desarrollo del proyecto, el cual se divide en coste de equipamiento, de *software*, de recursos humanos y coste total. Se presenta por tanto, el coste tanto de material empleado, como de los recursos humanos necesarios para desarrollar el proyecto.

A.1. Coste de equipamiento

El coste del equipamiento se divide en dos partes, una relativa al coste de los componentes del sistema sensor (Cuadro A.1) y otra al coste del equipo utilizado para el desarrollo (Cuadro A.2). Este equipo se trata de un ordenador portátil, cuyo coste asociado es relativo a la duración de este proyecto, teniendo un tiempo de vida estimado el equipo de cuatro años. Así, el total del coste del equipamiento necesario para la realización de este proyecto se detalla en el Cuadro A.3.

Para este proyecto, se ha optado por adquirir material para montar tres sistemas sensores con lo que realizar pruebas. Así, se han adquirido tres placas de desarrollo y tres sensores $MQ-135$.

Componente	**Cantidad**	**Precio/unidad**	**total**
ESP8266 ESP-12F WIFI	3	7,99 EUR	23,97 EUR
AZDelivery MQ-135B	3	3,36 EUR	10,08 EUR
Desglose del coste total Sensor	**3**	**11,35 EUR**	**34,05 EUR**

Cuadro A.1: Coste de los componentes del sistema sensor.

A.2. Coste de licencias software

En este capítulo se recoge el presupuesto requerido para adquirir las herramientas *software* requeridas para el desarrollo. El coste incurrido se expone en el Cuadro A.4. Tal y como se observa, la mayoría del software utilizado es de libre acceso, por lo que no incurre coste. Por otro lado, aunque el estudiante no ha tenido que adquirir la licencia de *Windows 10* (se incluía una licencia OEM en la compra del portatil) y *Overleaf Pro* (el

Componente	Precio
Ordenador portátil Lenovo ideapad 510 Intel Core i7-7500U 2.70GHz Memoria RAM DDR4 12GB Disco duro SSD 120 GB Tarjeta gráfica NVIDIA GeForce 940MX	800 EUR
Coste imputable (4 meses)	**66,7 EUR**

Cuadro A.2: Coste del equipo utilizado para el desarrollo.

Componente	Cantidad	Precio
Coste del sistema sistema sensor	3	34,05 EUR
Coste del equipo utilizado para el desarrollo	1	66,7 EUR
Desglose del coste total de equipamiento		**100,75 EUR**

Cuadro A.3: Coste total del equipamiento requerido.

tutor tenía acceso a una licencia), ambas se incluyen en el presupuesto para mostrar una versión realista del coste requerido.

Componente	Licencia	Número de licencias	Precio
Microsoft Windows 10	EULA	1	145,00 EUR
Microsoft Visual Studio Code	MIT	1	0 EUR
Kubernetes	Apache License v2.0	1	0 EUR
Docker	Apache License v2.0	1	0 EUR
Lens	EULA	1	0 EUR
Git	GPL	1	0 EUR
Overleaf Profesional 28 EUR/Mes	AGPLv3	4	112 EUR
Desglose del coste de software			**257 EUR**

Cuadro A.4: Coste de las licencias *software* requeridas.

A.3. Coste de recursos humanos

El coste de recursos humanos se relaciona con el número de horas empleadas por cada una de las figuras implicadas en el ciclo de desarrollo del proyecto. Como se espera, en este proyecto tan solo han participado dos personas, el estudiante y el tutor. Sin embargo, en este capítulo se ha querido mostrar el coste incurrido por cada una de las figuras habituales en un proyecto de desarrollo, según las horas requeridas por el estudiante y el tutor para el desarrollo del mismo. Este coste se encuentra detallado en el Cuadro A.5, donde cada figura está asociada con las siguientes responsabilidades:

- Jefe de proyecto. Encargado de la planificación y monitorización del sistema, además de la toma de las decisiones para la consecución de los objetivos del proyecto.
- Analista programador. Encargado del análisis y diseño de los módulos del sistema, además del análisis de las tecnología empleadas.

- Programador. Desarrollo de los módulos del sistema, la depuración del código y pruebas sobre el sistema.
- Secretaría. Incluye la redacción de la memoria y de los documentos adicionales sobre el desarrollo del sistema.

Concepto	**Horas**	**Coste/Hora**	**Coste total**
Jefe de proyecto	30	35	1.050 EUR
Analista Programador	35	22	770 EUR
Programador	200	16	3.200 EUR
Secretaría	100	12	1.200 EUR
Total			**6.220 EUR**

Cuadro A.5: Desglose del coste de recursos humanos.

A.4. Coste total

En este capítulo se detalla el coste total del proyecto como suma del coste de equipamiento, de licencias *software* y de recursos humanos. Además, se incluye un capítulo de costes generales relacionados con gastos de oficina y mantenimiento de las instalaciones. Este último capítulo supone un 15 % del coste de equipamiento, licencias y recursos humanos. Así, el coste total incurre en 7.564,41 euros, tal y como se observa en el Cuadro A.6.

Componente	**Precio**
Coste de equipamiento	100,75 EUR
Coste de licencias software	257 EUR
Coste de recursos humanos	6.220 EUR
Costes generales	986,66 EUR
Coste total del proyecto	**7.564,41 EUR**

Cuadro A.6: Desglose del coste total del proyecto.

Apéndice B

Anexo II: Manual de usuario

Este apartado de la memoria se proporciona una descripción para el uso del sistema desarrollado. Para ello, el primer lugar se detallará el procedimiento de despliegue de los componentes del sistema servidor. A continuación, se detallará el procedimiento de programación de un sistema sensor cualquier. Finalmente, se proporcionará un manual de uso de la interfaz de usuario acorde al rol del usuario.

B.1. Despliegue del sistema servidor

Los usuarios con el rol de administrador serán los encargados del despliegue del sistema servidor. Estos tienen que tener conocimientos previos de *Docker*, *Kubernetes* y *MongoDB*.

Tal y como se ha detallado previamente, para el despliegue en el servidor se necesitan los siguientes componentes:

- *Docker* [1] con la versión (v20.10.5).
- *Kubernetes* [2] con la versión (v1.19.7).
- *Lens* [3] con la versión (v1.19.7). *Lens* es un entorno de desarrollo integrado o IDE de código abierto, utilizado para controlar los clústeres de *Kubernetes*. Éste permite explorar y navegar los clústeres de *Kubernetes* sin necesidad de conocer los comandos de *kubectl*. Además, permite visualizar en tiempo real estadísticas, eventos, *pods*, servicios, configuraciones, errores y advertencias de *Kubernetes*.

Con *Docker*, *Kubernetes* y *Lens* instalados en el servidor, los pasos para el despliegue son los siguientes:

1. Desplegar en *Kubernetes* el componente base de datos a través del fichero de configuración *Yaml* correspondiente que hace el despliegue tanto de los *pods* como de los servicios necesarios para la puesta en marcha de la base de datos (véase la Sección 4.1.2). Los *Yaml*, que se encuentran en la carpeta del proyecto llamada *../Kubernetes/mongobd*, son los siguientes:
    - *mongodb-service.yaml*

[1] https://www.docker.com/products
[2] https://www.docker.com/products
[3] https://k8slens.dev/

- *mongodb-statefulset.yaml*

Los *Yaml* se despliegan ejecutando los siguientes comandos en una consola, siguiendo el orden en que aparecen los ficheros:

```
kubectl apply -f .\mongodb-service.yaml
kubectl apply -f .\mongodb-statefulset.yaml
```

Después de desplegar la base datos, el administrador tiene que insertar los usuarios del sistema y el rol de cada uno de ellos en la base de datos. Para ello, utilizando *Lens*, tiene que acceder al *pod* donde se ejecuta la base de datos, con el nombre de *mongodb-d-0*. Desde el *pod* se abre una *pod shell* y se acede a la base de datos. Para insertar un nuevo usuario se utiliza el siguiente comando:

```
db.getCollection('users').insert(
      {
          "id":x,
          "email":"xxx@xxx.xxx",
          "password":"xxx"
      }
  )
```

donde:

- id=1(Usuario Administrador) y id=2(Usuario Estándar).
- email= dirección de correo del usuario.
- password= Contraseña del usuario utilizando un Generador MD5 Hash.

2. Desplegar en *Kubernetes* el componente *Broker MQTT*. Tal y como se describió en la Sección *Broker* MQTT. Los ficheros de configuración Yaml son los siguientes:

   - *mosquitto-config.yaml*
   - *mosquitto-deployment.yaml*
   - *mosquitto-service.yaml*

   Los *Yaml*, que se encuentran en la carpeta *../Kubernetes/mosquitto*, se despliegan ejecutando los siguientes comandos en una consola siguiendo el orden en que aparecen los ficheros:

   ```
   kubectl apply -f .\mosquitto-config.yaml
   kubectl apply -f .\mosquitto-deployment.yaml
   kubectl apply -f .\mosquitto-service.yaml
   ```

3. Desplegar en *Kubernetes* el componente *Controlador*. En este caso, lo primero que se tendrá que realizar es la generación de la imagen de *Docker* correspondiente para su posterior despliegue en *Kubernetes* de la siguiente manera:

   a) Para generar esta imagen, se ejecutará el comando *docker build -t controlador:1.0.0 -f Dockerfile* desde la carpeta */kubernetes/controlador*:

b) Una vez generada la imagen, se ejecuta el *Yaml* del despliegue en *Kubernetes*, que se encuentra en */kubernetes/controlador*. El fichero es el siguiente:

- *controlador-statefulset.yam*l

4. Desplegar en *Kubernetes* el componente *Web Frontend*. Los pasos son los siguientes:

a) Desplegar el servidor web desde la carpeta */kubernetes/monitorizacion-servidorapp*. Para ello, se consideran los siguientes pasos:

1) Generación de la imagen de *Docker* con el siguiente comando:
   - *docker build t monitorizacionsweb*

2) Cargar la imagen de Docker en su registro de contenedores:
   - *docker run d p 5000:5000 restart=always name registry registry:2*
   - *docker tag monitorizacionsweb localhost:5000/monitorizacionsweb*
   - *docker push localhost:5000/monitorizacionsweb*

3) Ejecutar el *Yaml* del despliegue en *Kubernetes*:
   - *deployment.yaml*

b) Desplegar cliente web desde la carpeta */kubernetes/monitorizacion-clienteapp*. Para ello, se consideran los siguientes pasos:

1) Generación de la imagen de Docker con el siguiente comando
   - docker build t monitorizacioncweb

2) Cargar la imagen de Docker en su registro de contenedores
   - docker run d p 5001:5001 restart=always name registry registry:2
   - docker tag monitorizacioncweb localhost:5001/monitorizacioncweb
   - docker push localhost:5001/monitorizacioncweb

3) Ejecuta el YAML del despliegue en Kubernetes
   - deployment.yaml

B.2. Programación del sistema sensor

En esta sección se describe el proceso de programación de un sistema sensor cualquiera. Antes de realizar tal procedimiento, debe tenerse en cuenta que el sensor de $C0_2$ requiere de un tiempo de estabilización antes de proporcionar medidas válidas, por su diseño *hardware*. Así, el sistema sensor debe estar conectado a una toma de corriente durante al menos 24 horas antes de pasar a producción.

Para programar un sensor, se tienen que seguir los siguientes pasos:

1. Tener un ordenador con el entorno de desarrollo *Microsoft Visual Studio Code*[4].

2. Instalar en el *Visual Studio Code* la extensión de *PlatformIO*[5]. La Figura B.1 resume el procedimiento de instalación.

3. Con la extensión instalada, se tiene que cargar el código que se programará en el *NodeMCU*. La Figura B.2 muestra el *workspace* de *Visual Studio* con la estructura de los ficheros del *NodeMCU*.

[4] https://code.visualstudio.com/
[5] https://platformio.org/install/ide?install=vscode

Figura B.1: Instalación de la extensión *PlatformIO* en *Microsoft Visual Studio Code*.

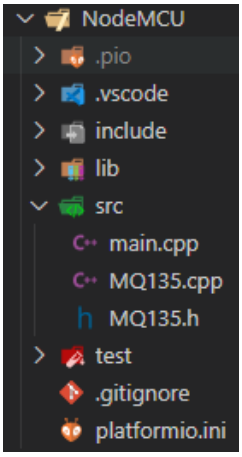

Figura B.2: Estructura de ficheros del código a programar en el *NodeMCU*.

4. Para programar el *NodeMCU* se tienen que seguir los siguientes pasos:

   a) El usuario administrador debe incorporar el nuevo sensor usando la interfaz de usuario web, tal y como se describirá en la Sección B.3.1.

   b) Una vez se haya añadido el nuevo sensor, se deben copiar los datos relativos al *Id Sensor* y *Espacio* (véase la Figura B.3), que son los identificadores del sensor y el espacio donde se desplegará, respectivamente.

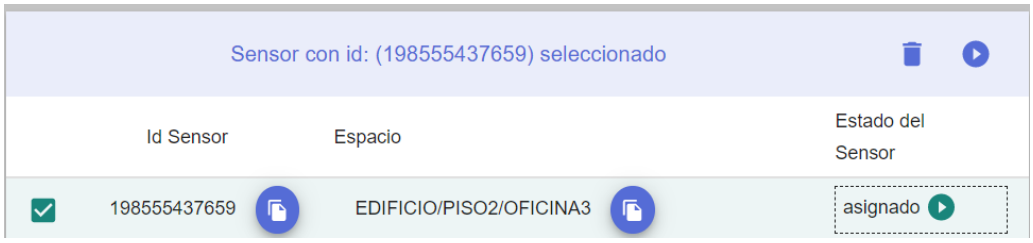

Figura B.3: Datos a copiar una vez se ha añadido el sensor al sistema.

   c) Además del *Id Sensor* y *Espacio*, se requiere de la siguiente información para poder programar convenientemente el sensor:

   - Nombre o *ssid* de la red WiFi a la que tendrá acceso el sensor.
   - Contraseña de la red WiFi.

- La dirección IP del sistema servidor.

d) Una vez recolectada la información sobre estos cinco campos, se debe modificar el fichero *main.ccp*, localizado en la estructura mostrada previamente (véase la Figura B.2), tal y como se muestra en la Figura B.4. En ese código las variables *ID*, *ssid*, *password* y *mqtt_server* hacen referencia a la información relativa al identificador del sensor, el nombre de la red WiFi, la contraseña de la red WiFi y la dirección IP del sistema servidor. Por otro lado, la definición *MQTT_TOPIC* hace referencia a la ubicación o tópico del sensor.

```
#define MQTT_TOPIC "xxxxxxxx"
String ID = "xxxxxxx";
const char *ssid = "xxxxxxx";
const char *password = "xxxxxx";
const char *mqtt_server = "xxxxx";
```

Figura B.4: Configuración de los parámetros del sensor en el fichero *main.ccp*.

e) Una vez editado el fichero main.ccp, se conecta el módulo sensor al ordenador vía USB como se ve en la Figura B.5

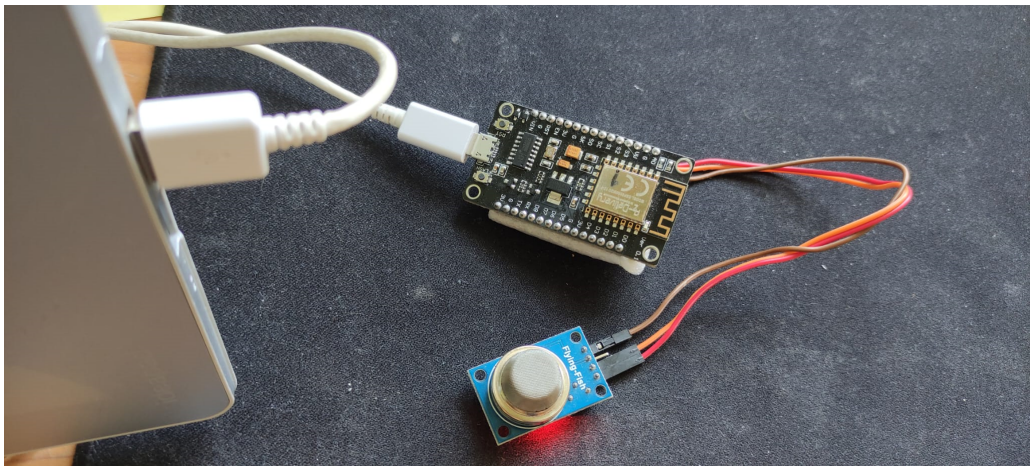

Figura B.5: Conexión vía USB del sistema sensor con el ordenador.

f) El último paso consiste en cargar el código en el *firmware* del *NodeMCU*. Para ello, se utiliza el botón el botón situado en la parte inferior izquierda del *Visual Studio Code* y que es proporcionado por *PlatformIO*. Dicho botón aparece remarcado en la Figura B.6.

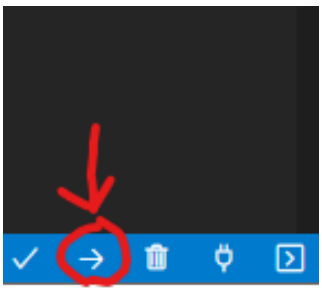

Figura B.6: Botón que permite cargar el código en el *firmware* del *NodeMCU*.

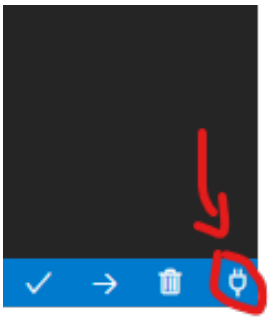

Figura B.7: Icono de acceso a la aplicación *Serial Monitor*.

En caso de que un sensor presente un funcionamiento errático durante producción, entrando en modo *Error sensor*, se debe verificar su funcionamiento acorde a los siguientes pasos:

1. Conectar el módulo sensor al ordenador vía USB, para a continuación, lanzar el programa *Serial Monitor* proporcionado por *PlatformIO*, que permite realizar una conexión con el sensor, para así poder visualizar mensajes sobre su estado en una consola. Véase en la Figura B.7 en icono de acceso a tal aplicación dentro de *Visual Studio Code*.

2. En esta consola, se pueden comprobar dos situaciones.

   - Problema relacionado con la conexión a la red WiFi, como por ejemplo se muestra en la Figura B.8.

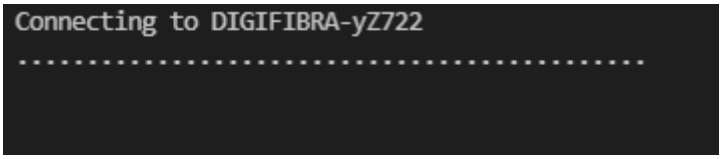

Figura B.8: Problema relacionado con la conexión a la red WiFi.

   - Problema relacionado con la conexión al *Broker MQTT*, como por ejemplo se muestra en la Figura B.9.

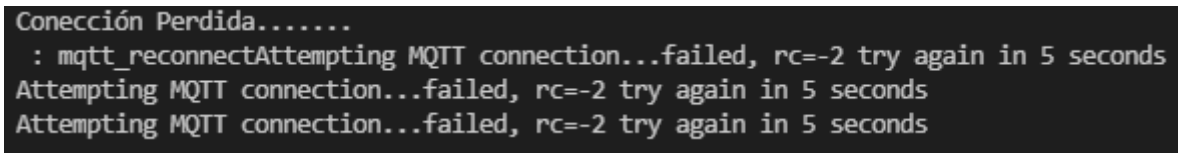

Figura B.9: Problema relacionado con la conexión al *Broker MQTT*.

Nótese que si, estando el sensor funcionando correctamente se accede a la consola descrita previamente. Entonces, se mostrarán mensajes acerca de la recolección de datos de CO_2 y su posterior envío al *Broker MQTT*, tal y como se observa en la Figura B.10.

```
..........
WiFi connected
IP address:
192.168.1.137
Publish message: --> Valor de RZero: 24408.46
--> Valor de RZero: 24408.46
--> Valor de RZero: 24408.46
--> Valor de RZero: 24408.46
--> Valor de RZero: 24408.46
--> Valor de RZero: 24408.46
--> Valor de la Media de RZero: 24408.46
Publish message: {"id":"295152264538", "dato":"390.80"}
Publish message: {"id":"295152264538", "dato":"390.80"}
```

Figura B.10: Información proporcionada por la consola durante un correcto funcionamiento del sistema sensor.

B.3. Manual de uso del entorno web

El *frontend* puede ser utilizado tanto por usuarios con privilegios de administrador como por usuarios estándar. Los usuarios, dependiendo de los privilegios que tengan, accederán a funcionalidades propias de su Rol.

El acceso al *frontend* se realiza a través de una página de *login* tal y como se muestra en la Figura B.11. En esta pantalla, el usuario introduce un identificador y una contraseña válidas, que deben haber sido previamente registradas en el sistema por el administrador. Una vez autenticado, el usuario accede al portal que le corresponde según su rol.

Figura B.11: Pantalla de *login* del *Web Frontend*.

En caso de que haya algún tipo de error en relación a los datos introducidos, la interfaz emite una notificación a través de un mensaje, el cual saldrá en la parte inferior izquierda de la pantalla. Los distintos tipos de error aparecen en la Figura B.12.

En caso de que el usuario quiera cerrar sesión, sin importar su rol, tiene que hacer *click* en el icono de usuario localizado en la esquina superior derecha en la pantalla del menú principal. Luego, en la ventana que se despliega, se cierra sesión haciendo *click* en el botón (CERRAR SESIÓN), redirigiéndole de nuevo a la pantalla de *login*. Véase la Figura B.13.

Figura B.12: Mensajes de error del proceso de *login*.

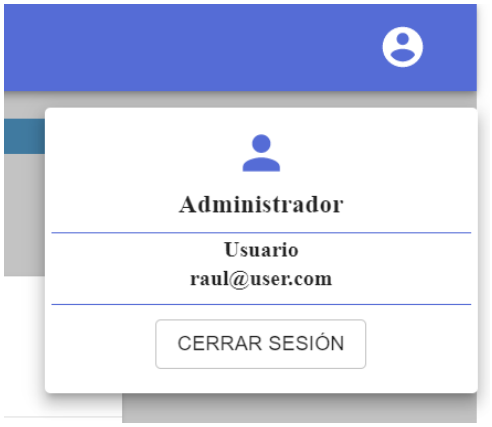

Figura B.13: Cierre de sesión.

A continuación, se describen las diferentes acciones que pueden llevar a cabo los usuarios administradores y los usuarios estándar.

B.3.1. Usuario con rol administrador

Los usuarios administradores tienen acceso al menú mostrado en la Figura B.14, que permite visualizar la pantalla de sensores y la de espacios.

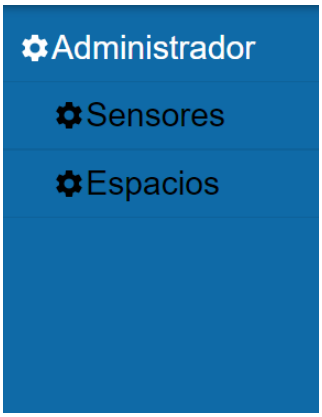

Figura B.14: Menú principal del usuario administrador.

B.3.1.1. Pantalla de espacios

Esta pantalla, que se muestra en la Figura B.15, permite añadir o eliminar los espacios donde se desplegarán los sensores, además de poder visualizar la tabla con todos los espacios del sistema y un menú desplegable(véase la Figura B.16) donde se ven los

diferentes niveles desde el principal hasta cada uno de los niveles y subniveles de cada espacio. En la tabla de espacios se puede ordenar cada columna de mayor a menor y viceversa.

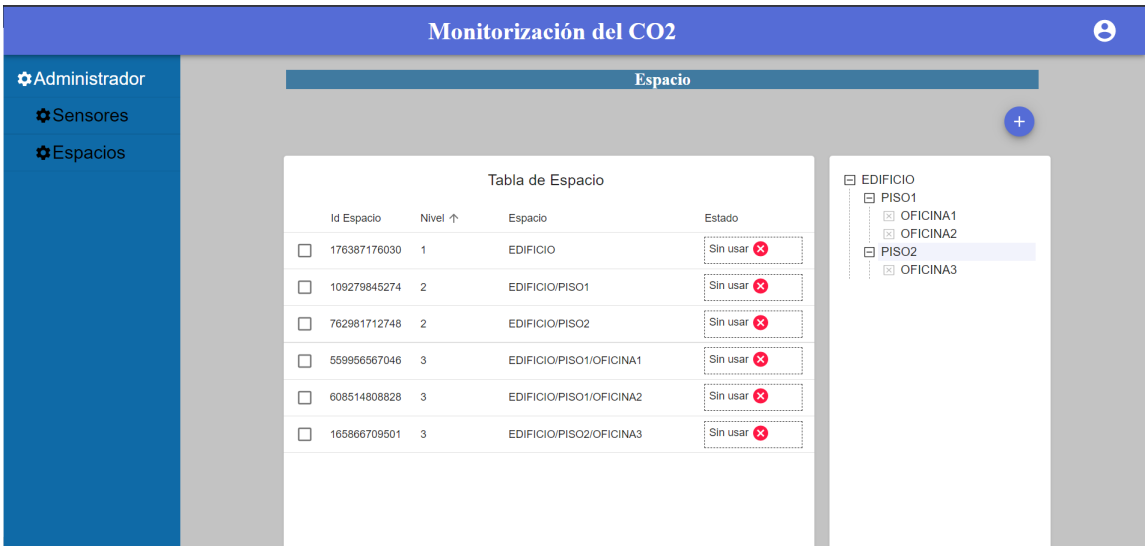

Figura B.15: Pantalla de espacios.

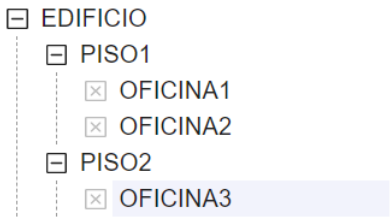

Figura B.16: Visualización de los diferentes niveles y espacios disponibles.

En la tabla de espacios, el estado de estos se muestra con un icono en verde, en caso de que estén siendo utilizados por un sensor y con un icono en rojo en caso contrario. Véase un ejemplo de esta notación en la Figura B.17.

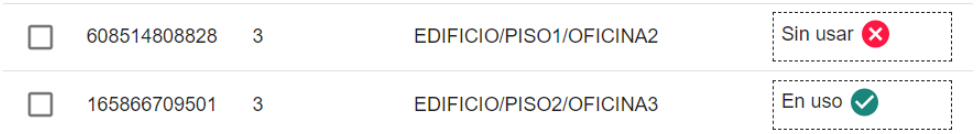

Figura B.17: Ejemplo de la notación utilizada en la tabla de espacios.

Para añadir un espacio nuevo, se tiene que hacer *click* en el botón (+), tal y como se ve en la Figura B.18. Después de hacer *click*, se despliega una ventana donde aparece el formulario (véase la Figura B.19) de inserción de un espacio nuevo. En este formulario, el usuario tiene que completar todos los campos, ya que todos son obligatorios. Cuando todos los campos son correctos, se guardan haciendo *click* en el botón (Guardar) o en caso de que se quiera cancelar se hace *click* en el botón (Cancelar).

Figura B.18: Botón añadir espacio.

Figura B.19: Ventana añadir Espacio.

En caso de que haya algún tipo de error al intentar guardar el formulario, la interfaz lo notifica a través de un mensaje emergente que saldrá en la parte inferior izquierda de la pantalla. Además, si alguno de los campos no se completa aparecerá en rojo el contorno del campo (véase la Figura B.20). Cuando se guardan los datos de forma correcta, la interfaz lo notifica igualmente con un mensaje emergente que saldrá en la parte inferior izquierda (véase la Figura B.21).

Figura B.20: Mensajes de error al añadir un espacio nuevo.

Figura B.21: Mensaje que confirma la creación del espacio correctamente.

Si se quiere eliminar un espacio, se selecciona éste como se ve en la Figura B.22. Al seleccionar el espacio a eliminar se muestra el nombre y un botón de eliminar en la esquina superior derecha de la tabla. Para poder eliminar un espacio, éste no puede tener espacios dependientes de él, además el estado de éste tiene que ser igual a *sin usar*. En el caso de no cumplir estas reglas, la interfaz notifica a través de un mensaje emergente que saldrá en la parte inferior izquierda (véase la Figura B.23). En el caso de cumplir con las especificaciones al eliminar el espacio, la interfaz notifica a través de un mensaje emergente que saldrá en la parte inferior izquierda, que la eliminación se llevó a cabo correctamente (véase la Figura B.24).

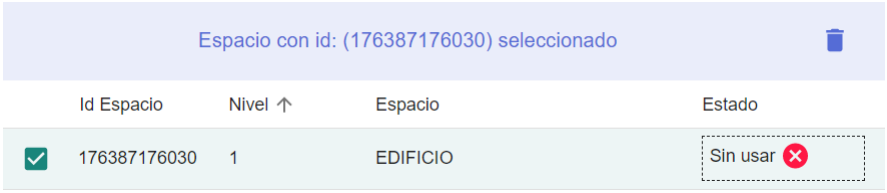

Figura B.22: Eliminar un espacio.

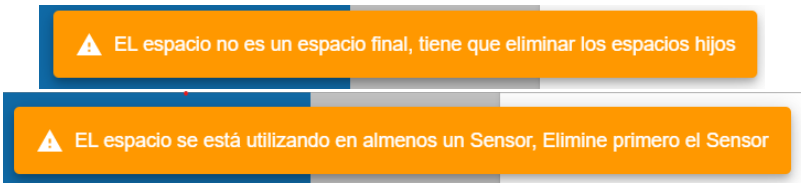

Figura B.23: Mensajes de advertencia al intentar eliminar un espacio.

Figura B.24: Mensaje de espacio eliminado correctamente.

B.3.1.2. Pantalla de sensores

Esta pantalla, véase la Figura B.25, permite visualizar todos los sensores del sistema, insertar y eliminar sensores, así como cambiarlos de modo.

Después de registrarse el usuario, se muestra directamente la ventana de sensores que es la principal. El usuario, en caso de actualizar la página manualmente en el navegador web, siempre será redirigido hasta esta ventana.Esta pantalla se actualiza cada 10 segundos automáticamente, por lo que los cambios realizados no se verán hasta pasar este tiempo.

Figura B.25: Pantalla de sensores.

En esta pantalla, la tabla de sensores permite visualizar todos los sensores en el sistema. Cada columna se puede ordenar de menor a mayor y viceversa. En cada fila, como se puede ver en la Figura B.26, si se hace *click* en los iconos en azul que se encuentran al lado del valor de *Id Sensor* y *Espacio*, permiten copiar estos valores en el portapapeles del sistema operativo, con el propósito de facilitar la programación de sensor, tal y como se verá en la sección de programación del sistema sensor vista antes. En esta tabla además se puede ver el estado del sensor (véase la Figura B.27). Esto es, si está en *asignado*, significa que el sensor fue creado y asignado a un espacio, *ejecutando* significa que el sensor se está ejecutando correctamente y *Error sensor* significa que el sensor no se está ejecutando. Las razones para este ultimo estado incluyen que el sensor físicamente no esté conectado a una toma de corriente, que haya perdido la conexión a la red wifi o que el módulo del servidor que gestiona los sensores esté caído. En caso de que el estado sea igual *Error sensor*, el administrador desde esta tabla puede volver a ponerlo en *ejecutando*, una vez haya resuelto el problema existente.

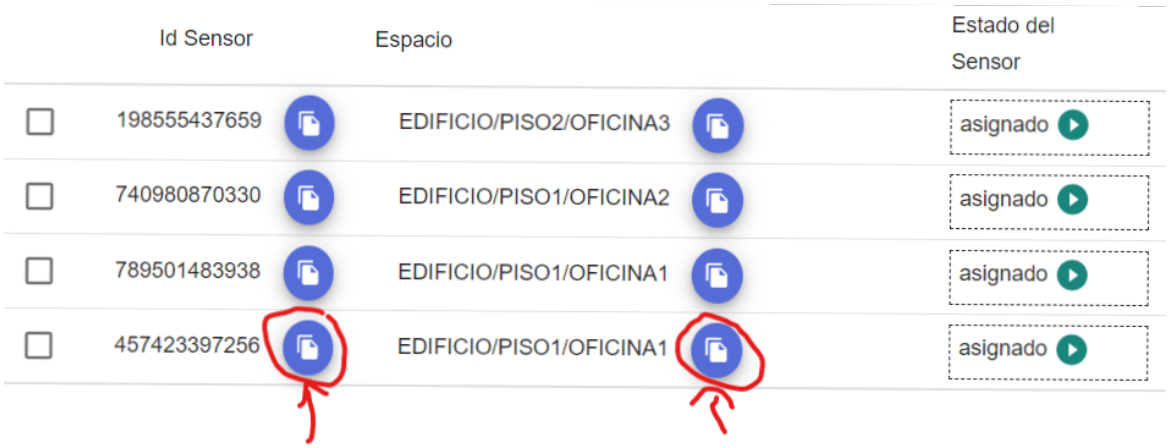

Figura B.26: Tabla de sensores.

Figura B.27: Posibles estados de los sensores.

Para añadir un sensor nuevo se tiene que hacer *click* en el botón (+) de la pantalla de sensores (véase la Figura B.25). Después de hacer *click*, se despliega una ventana (véase la Figura B.28) donde aparece el formulario de inserción de un sensor nuevo. El administrador tiene que completar todos los campos, ya que todos son obligatorios. Cuando todos los campos son correctos, se guardan haciendo *click* en el botón (Guardar) o en caso de que se quiera cancelar se hace *click* en el botón (Cancelar). En el caso de que ocurra algún tipo de error al intentar guardar el formulario, la interfaz lo notifica a través de un mensaje emergente (véase la Figura B.29) que saldrá en la parte inferior izquierda, además si alguno de los campos no se completa, aparecerá en rojo el contorno del campo. En caso de que no hubiera ningún error, la interfaz lo notifica igualmente, tal y como se observa en la Figura B.30.

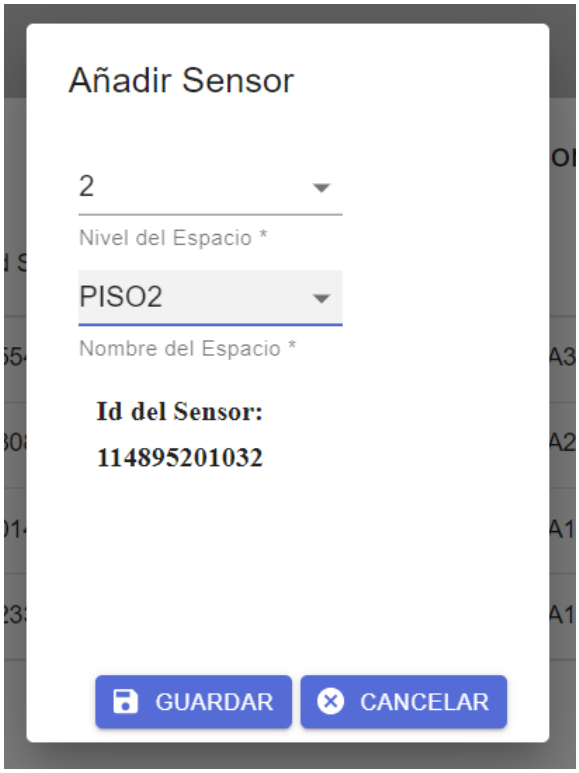

Figura B.28: Ventana añadir sensor

Figura B.29: Mensajes de error al añadir un sensor nuevo.

Figura B.30: Mensaje que indica que el sensor se creó correctamente.

Si se quiere eliminar un sensor, éste debe seleccionarse en la tabla de sensores, tal y como se muestra en la Figura B.31. Al seleccionarlo, se mostrará el botón de eliminar en la esquina superior derecha de la tabla. Después de seleccionar el sensor a eliminar, se hace *click* en el botón de eliminar. Entonces, si el estado del sensor es *asignado*, éste se elimina directamente y si el estado del sensor es *ejecutando*, el proceso se demora unos instantes, ya que el sistema tiene que detener los procesos que tiene corriendo en relación a tal sensor. Los mensajes relacionados con este procedimiento de eliminación se muestran en la Figura B.32.

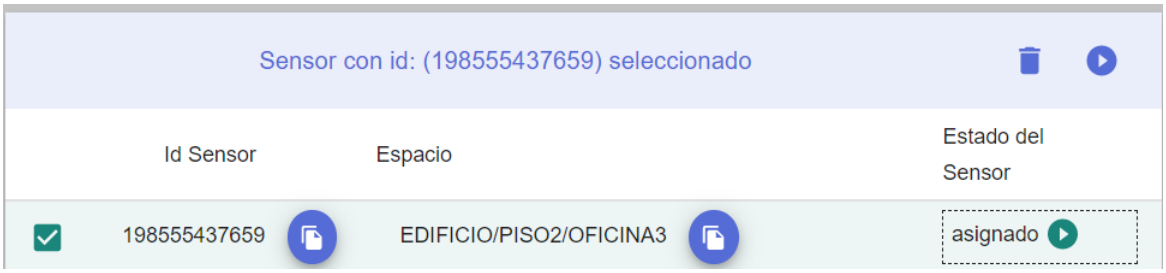

Figura B.31: Eliminar un sensor.

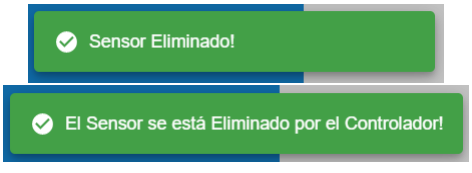

Figura B.32: Mensajes mostrados al eliminar un sensor

Si se quiere poner *en ejecución* un sensor, en primer lugar debe seleccionarse en la tabla de sensores, tal y como se hizo en el proceso de eliminación descrito previamente. Entonces, se mostrará un botón de *ejecutar*, que permitirá cambiar a tal estado. Este botón solo se muestra si el estado del sensor es *asignado* o *Error sensor*. Téngase en cuanta que si un sensor se encuentra en estado *error*, antes de pasar a modo *ejecución*, éste debe revisarse siguiendo la metodología descrita en la Sección de programación del sistema sensor vista antes.

B.3.2. Usuario con rol estándar

Los usuarios con rol estándar tienen acceso al menú mostrado en la Figura B.33, que permite visualizar la pantalla de inicio y la de espacios. Además, justo debajo de las opciones del menú, se puede ver en todo momento el estado general de los espacios. Así, si un espacio supera los niveles de CO_2 permitidos, aparecerá una alerta aquí, tal y como se muestra en la Figura B.34.

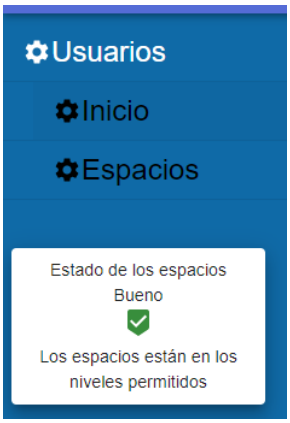

Figura B.33: Menú principal del usuario estándar

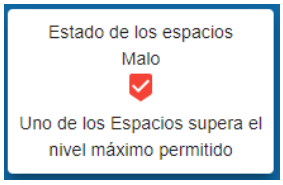

Figura B.34: Mensaje de alerta en el menú de usuario estándar.

B.3.2.1. Pantalla de espacios

En esta pantalla se visualizan todos los espacios del sistema, como se puede ver en la Figura B.35. Para seguir los cambios en la concentración de CO_2 en los espacios, se muestran dos tablas: la principal con todos los espacios del sistema y otra, donde solo se encuentran los espacios que superan los límites permitidos. Además, en la parte derecha de la pantalla, se puede ver un cuadro que muestra el espacio con la media de concentración de CO_2 más alta de todo el sistema. Los niveles de CO_2 mostrados se acompañan con un código de colores, según el criterio descrito previamente en la Sección 2.2. Así, un nivel bueno <= 500ppm se muestra en verde, un nivel medio 500 <ppm<= 800 en naranja y

un nivel malo > 800ppm en rojo. En esta pantalla, la tabla de los espacios se divide en 4 columnas, mostrando el nombre de los espacios, el *ID* del sensor que tiene asociado el espacio, el nivel de CO_2 en ppm y la hora de la última actualización. La tabla de espacios con niveles de CO_2 que superan los límites permitidos, solo muestra aquellos espacios que tengan como media entre los sensores que tiene asignado un número mayor a 800 ppm, es decir, un nivel malo.

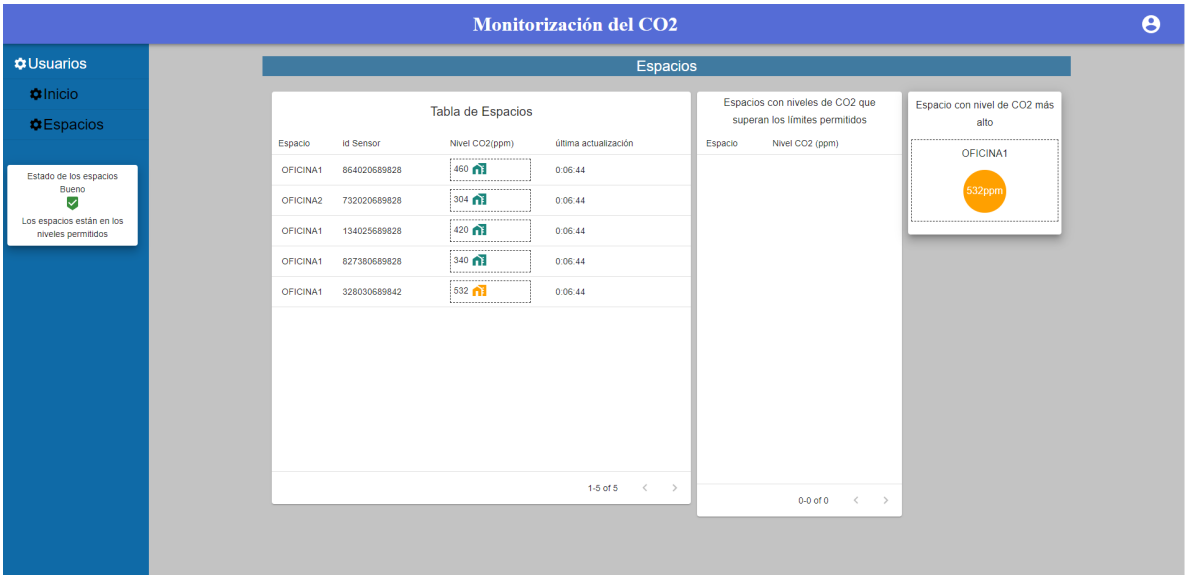

Figura B.35: Pantalla de espacios en el usuario con rol estándar.

B.3.2.2. Pantalla de inicio

Esta pantalla, la cual es la primera que visualiza el usuario después del *login* o después de recargar la página en el navegador, muestra la media general de concentración de CO_2, así como un gráfico que muestra la evolución de la concentración por horas, el cual es configurable mediante filtros de búsqueda. Véase la Figura B.36.

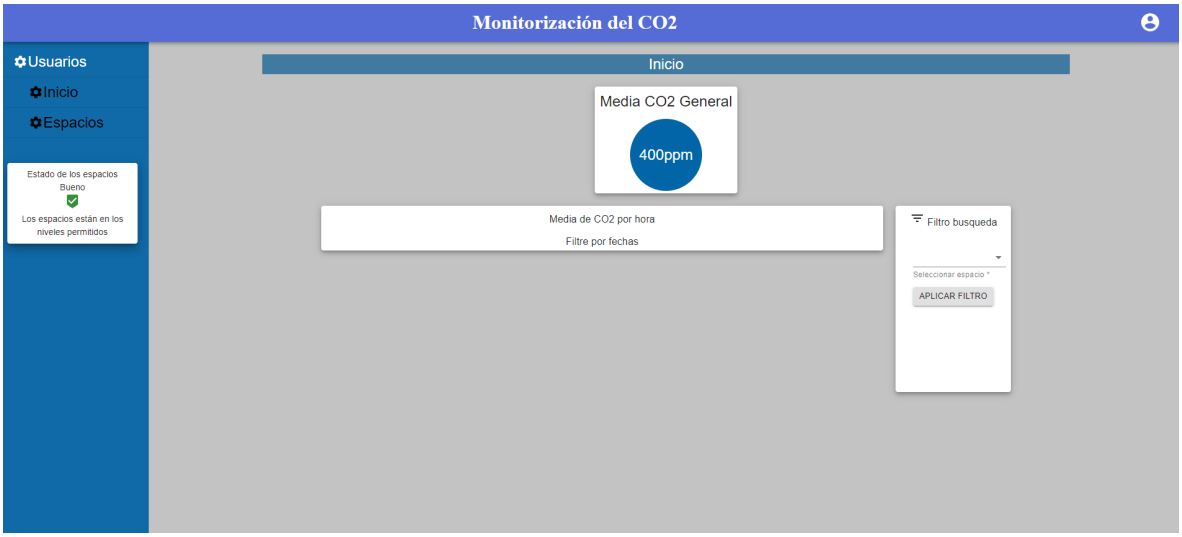

Figura B.36: Pantalla de inicio del usuario con rol estándar.

Aplicando filtros, como se puede ver en la Figura B.37, se puede ver la media de CO_2 por horas del espacio seleccionado. En el caso que el espacio tenga más de un sensor asociado, aparecerá en el gráfico cada sensor con un color diferente, pudiendo seleccionar uno o varios de los sensores, para que aparezca solo en el gráfico. Para ejecutar el filtro basta con seleccionar el espacio y la fecha y hacer *click* en el botón de aplicar filtro.

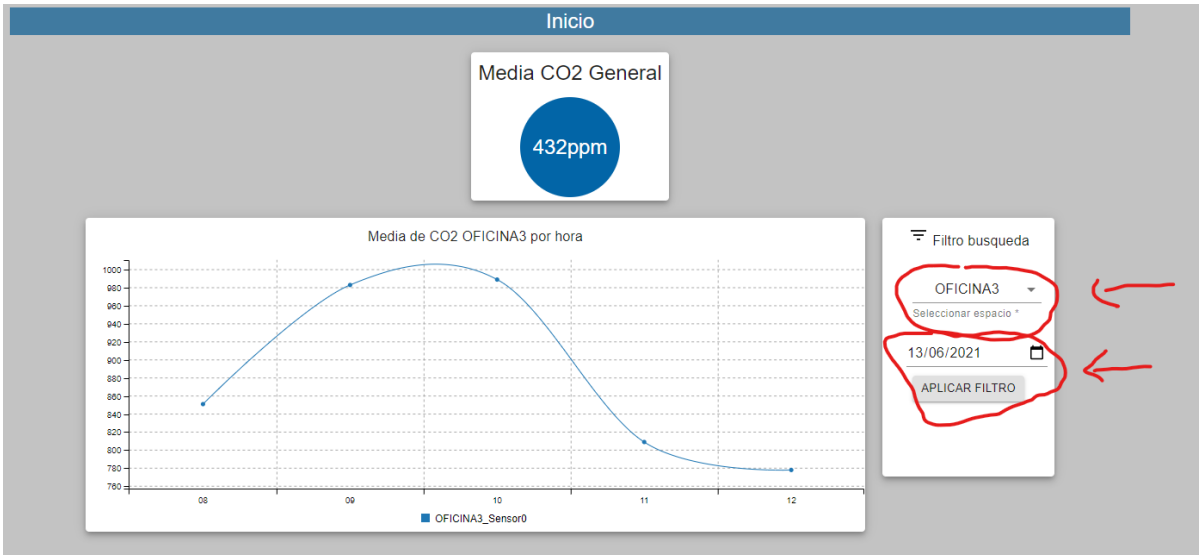

Figura B.37: Pantalla de inicio con un filtro configurado.

Al aplicar el filtro, puede que no haya registros para la fecha y/o ubicación seleccionada. En tal situación, aparecerá una advertencia en un mensaje emergente que saldrá en la parte inferior izquierda (véase la Figura B.38) . Además si no se completa los dos datos obligatorios, no se aplicara el filtro, marcando en rojo el contorno del campo no definido.

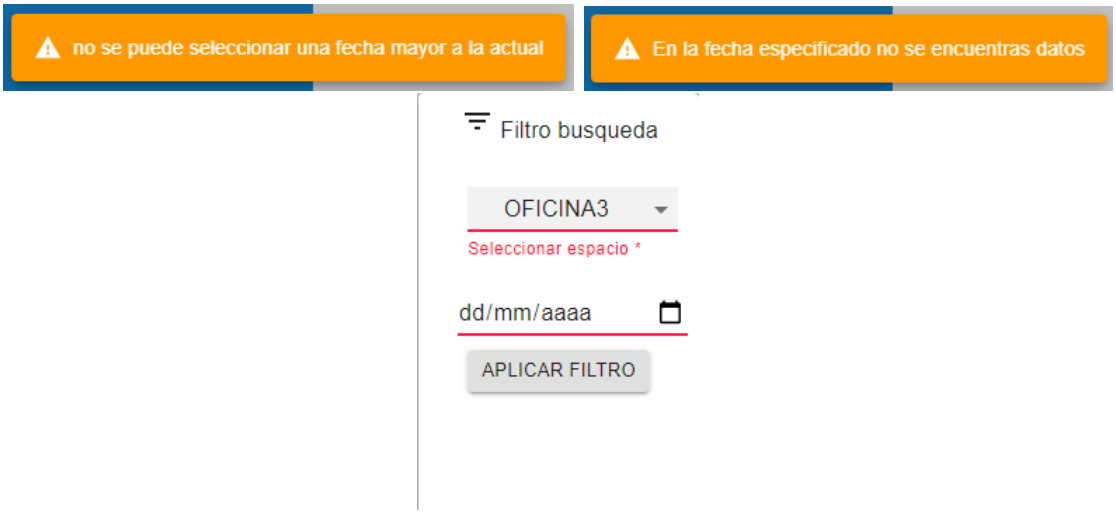

Figura B.38: Advertencias al aplicar filtros en la Pantalla de inicio.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA SUPERIOR

Universidad de Alcalá