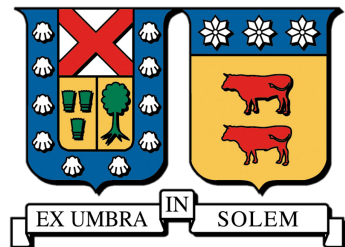


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE



**SERVICIOS DE IOT DE AMAZON WEB
SERVICES APLICADOS A UN SISTEMA DE
MONITOREO A TIEMPO REAL DE
ESTACIONAMIENTOS**

ESTEBAN IGNACIO MOHR FESTER
MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL ELECTRÓNICO MENCIÓN COMPUTADORES

PROFESOR GUÍA: DR AGUSTIN JOSÉ GONZÁLEZ VALENZUELA
PROFESOR CORREFERENTE: RUDY MALONNEK WETZEL

VALPARAÍSO, MARZO, 2023

Agradecimientos

Al profesor Agustín González por su patrocinio y apoyo durante mi proyecto de memoria.

Al profesor Rudy Malonnek por participar de mi proyecto como profesor co-referente.

A mi amigo de la infancia Juan Ignacio Ruhe por diseñar un modelo en 3D de una carcasa protectora para los sensores ultrasonido.

A mi novia Isidora Indo por brindarme asistencia durante las pruebas del sistema.

A mi familia, mis amigos de la infancia y mi novia, quienes confiaron que lo lograría

Servicios de IOT de Amazon Web Services aplicados a un sistema de monitoreo a tiempo real de estacionamientos

Esteban Mohr

Memoria para Optar al Título de Ingeniero Civil Electrónico, Mención Computadores.

Profesor Guía: Dr. Agustín González

Marzo 2021

Resumen

Una tecnología altamente usada en la denominada *Industria 4.0* o *Cuarta Revolución Industrial* es el *Internet de las Cosas* (IoT por sus siglas en inglés) la cual permite comunicar en tiempo real dispositivos utilizados tanto en la industria como en el hogar. Es por esta que grandes compañías como Amazon por medio de su plataforma de computación en la nube “Amazon Web Services” ofrece servicios que facilitan el desarrollo de aplicaciones que hacen uso de esta tecnología. Este proyecto tiene como objetivo explorar las funcionalidades de estos servicios a través de la creación de un prototipo de un caso de uso, el cual consiste en un sistema de monitoreo de puestos de estacionamiento. El diseño de este sistema se compone de cuatro subsistemas y su implementación integra elementos tales como un microcontrolador “ESP32 PICO” de pequeñas dimensiones y bajo costo; el sistema operativo de tiempo real FreeRTOS que facilita el desarrollo del código fuente del microcontrolador; sensores ultrasonido para detectar proximidad; protocolo de mensajería MQTT y una aplicación móvil para dispositivos “Android”. Por medio de pruebas experimentales definidas para comprobar los requerimientos de cada uno de los subsistemas y el sistema completo se logra comprobar su correcto funcionamiento y queda demostrado que utilizando componentes simples y de bajo costo es posible crear un sistema que haciendo uso del Internet de las Cosas satisfaga una necesidad de la vida diaria. Finalmente se analizan las posibles mejoras que puede presentar el sistema y cómo escalar el sistema para ser convertido en un sistema que pueda ser utilizado en un entorno real.

Palabras clave: Internet de las Cosas, Amazon Web Services, Estacionamiento Inteligente, ESP32, FreeRTOS, MQTT

Amazon Web Services IoT Services applied to a real-time parking monitoring system.

Esteban Mohr

Final Project Report towards the fulfillment of the Electronic Civil Engineering, Major in Computers Degree.

Advisor: Agustín González

March 2021

Abstract

A highly used technology in the so-called *Industry 4.0* or *Fourth Industrial Revolution* is the *Internet of Things (IoT)*, which allows devices used both in industry and at home to communicate in real time. This is why big companies like Amazon through their cloud computing platform “Amazon Web Services” offer services that facilitate the development of applications that make use of this technology. This project aims to explore the functionalities of these services through the creation of a use case prototype, which consists of a parking spot monitoring system. The design of this system consists of four subsystems and its implementation integrates elements such as a small and low-cost microcontroller “ESP32 PICO”; the real-time operating system FreeRTOS that facilitates the development of the microcontroller source code; ultrasonic sensors to detect proximity; MQTT messaging protocol and a mobile application for “Android” devices. Through experimental tests defined to verify the requirements of each of the subsystems and the entire system, its correct operation is verified and it is demonstrated that by using simple and low-cost components, it is possible to create a system that makes use of the Internet of Things to satisfy a daily life need. Finally, the possible improvements that the system can present and how to scale the system to be converted into a system that can be used in a real environment are analyzed.

Keywords: Internet of Things, Amazon Web Services, Smart Parking, ESP32, FreeRTOS, MQTT

Glosario

AWS Amazon Web Services. 9, 16, 17, 23, 28, 31, 33, 36, 44, 45, 47–49, 51, 53–58, 75–77

CLP Pesos Chilenos. 19, 21

esp-idf Espressif IoT Development Framework. 22, 25

IAM Identity and Access Management. 36, 37

IIoT Industrial Internet of Things. 1

IoT Internet of Things. 1–3, 6, 8, 12, 16, 23, 33, 34, 44, 46, 47, 53, 55–58, 75

kB Kilobytes. 21

MB Megabytes. 21

MQTT Message Queuing Telemetry Transport. 15, 23, 44, 75, 77

SMPE Sistema de monitoreo de puestos de estacionamiento. 9, 13, 18, 20, 53, 54

Índice general

Agradecimientos	I
Resumen	II
Abstract	III
Índice de figuras	VIII
Índice de tablas	0
1. Introducción	1
1.1. Antecedentes	1
1.2. Definición del problema	2
1.3. Objetivos	3
1.4. Alcance	3
1.5. Protocolo de comunicación MQTT	4
1.6. Proyectos existentes de Smart Parking	5
1.6.1. Sistema inteligente de estacionamientos desarrollado por Mo- bidev	5
1.6.2. Sistema de Smart Parking desarrollado por Shoban Kumar . . .	7
1.7. Herramientas utilizadas	9
1.8. Resumen de capítulos siguientes	9
2. Diseño del SMPE	11
2.1. Arquitectura del sistema	11
2.1.1. Subsistema de Sensores	12
2.1.2. Subsistema de dispositivos de borde	13
2.1.3. Subsistema de Nube de IoT	14

2.1.4.	Subsistema de Aplicación	15
2.2.	Requisitos	15
2.2.1.	Requisitos funcionales	15
2.2.2.	Requisitos no funcionales	16
2.3.	Requisitos de Subsistemas de SMPE	16
2.3.1.	Subsistema de Sensores	16
2.3.2.	Subsistema de Dispositivos de Borde	16
2.3.3.	Subsistema de Nube de IoT	17
2.3.4.	Subsistema de Aplicación	17
3.	Implementación	18
3.1.	Componentes utilizados	18
3.1.1.	Subsistema de Sensores	18
3.1.2.	Subsistema de Dispositivos de Borde	20
3.1.3.	Subsistema de Nube de IoT	23
3.1.4.	Subsistema de Aplicación	23
3.2.	Desarrollo de los submódulos	24
3.2.1.	Conexión de los sensores	24
3.2.2.	Instalación y configuración de esp-idf	25
3.2.3.	Script desarrollado para microcontrolador	28
3.2.4.	Configuración de nube de AWS	33
3.2.5.	Desarrollo de aplicación de usuario	37
4.	Pruebas y resultados	40
4.1.	Funcionamiento de los sensores	40
4.2.	Comunicación entre Microcontrolador AWS	44
4.3.	Comunicación entre AWS y aplicación Android	47
4.4.	Funcionamiento del sistema completo	50
4.5.	Medidas de desempeño	51
5.	Conclusiones y trabajo futuro	53
	Bibliografía	55
	Anexos	61
A.	Código fuente de smart_parking.c	61

B. Código fuente de PubSubActivity.java	70
C. Explicación detallada del código fuente	74
D. Diseño en 3D de carcasa protectora para sensores ultrasonido	79

Índice de figuras

1.1. Ejemplo de comunicación MQTT	4
1.2. Arquitectura del sistema de Smart Parking desarrollado por Mobidev . .	6
1.3. Detección del estado de los puestos de estacionamientos	7
1.4. Arquitectura del sistema desarrollado por Shoban Kumar	8
1.5. Vistas de la pantalla de la Raspberry y de administrador	9
2.1. Diagrama de contexto del SMPE	11
2.2. Diagrama de bloques de subsistemas del SMPE	12
2.3. Funcionamiento del Software del dispositivo de borde	14
3.1. Diagrama de bloques de la implementación del SMPE	18
3.2. Sensor ultrasonido HY-SRF05 [24]	19
3.3. Distribución de pins de sensor ultrasonido HY-SRF05	20
3.4. Dispositivo ESP32-PICO-KIT V4, [22]	21
3.5. Distribución de pines del ESP32-PICO-KIT, [22]	22
3.6. Diagrama de conexiones de los sensores ultrasonido al microcontrolador	24
3.7. Fotografía de conjunto de sensores conectados al Dispositivo de Borde .	25
3.8. Ejemplo de ejecución del ambiente ESP-IDF Powershell	27
3.9. Código de función <i>app_main</i>	30
3.10. Creación de política haciendo uso del creador	34
3.11. Creación de política haciendo uso del editor JSON	35
3.12. Creación de objeto finalizado	35
3.13. Roles creados en servicio “Amazon Cognito” observados en el servicio IAM	37
3.14. Política asociada al rol exitosamente	37
3.15. Diseño de interfaz gráfica de aplicación de usuario, observado desde “Android.Studio”	38

3.16. Diagrama de flujo de código del archivo PubSubActivity.java	39
4.1. Entorno de pruebas para sensores y microcontrolador	42
4.2. Ejemplo de mediciones de los sensores mostrados en la línea de comandos	43
4.3. Mensajes del microcontrolador recibidos por el cliente de prueba de AWS	45
4.4. Comportamiento de la aplicación al recibir mensajes del cliente de prueba de AWS	48
4.5. Comportamiento de la aplicación al recibir mensajes del cliente de prueba de AWS con el dispositivo móvil conectado a otra red.	49
D.1. Modelo 3D de protección para sensores infrarojos	79
D.2. Plano de vistas de protección para sensores infrarojos	80

Índice de tablas

4.1. Resultados de la prueba de medición de los sensores	42
4.2. Resultados de mediciones de desempeño	52

Capítulo 1

Introducción

1.1. Antecedentes

La continua mejora de las velocidades de transmisión de datos a través de Internet ha abierto la puerta a nuevas posibilidades para la industria, ya que estas permiten transmitir información valiosa en tiempo real, lo cual está cambiando la manera en que opera la industria. Es posible afirmar que actualmente nos encontramos en la cuarta revolución industrial, también conocida como la *Industria 4.0*. Esta revolución tecnológica engloba diferentes tecnologías como la robótica, la inteligencia artificial, las tecnologías cognitivas, la nanotecnologías, y el Internet de las cosas (“Internet of Things” o IoT) [1].

El IoT consiste en la tendencia tecnológica de conectar los objetos del mundo cotidiano a la red, de tal manera que puedan intercambiar información y crear aplicaciones que generen valor a través de ella. Un artículo de Oracle lo define de la siguiente manera: “La Internet de las cosas (IoT) describe la red de objetos físicos (“cosas”) que llevan incorporados sensores, software y otras tecnologías con el fin de conectarse e intercambiar datos con otros dispositivos y sistemas a través de Internet.” [2] Esto tiene aplicaciones en diversas áreas, tanto para el hogar (domótica) como en la industria. La aplicación de IoT en la industria es denominado también como “Industrial Internet of Things”(IIoT) [3].

El aumento del uso de esta tecnología ha hecho que grandes compañías como Amazon, Microsoft, y Google hayan creado servicios integrados en la nube para facilitar la

creación de aplicaciones de IoT [5] [7] [8]. Es por esto que resulta interesante explorar las posibilidades que este tipo de herramientas entrega y su potencial a través del desarrollo de un caso de uso de estos servicios.

1.2. Definición del problema

El continuo crecimiento de la población y las zonas urbanas traen consigo diferentes problemáticas, entre ellas el complejo control del tráfico automovilístico. Según un sondeo realizado por Cadem, en Chile el 71 % de los hogares declara poseer al menos un automóvil, según este estudio el 2020 se comercializaron 415.581 vehículos nuevos siendo este el segundo año con mayor ventas en la historia [9]. Es por esto que una situación común a la cual se enfrentan los conductores es encontrar un sitio para aparcar sus vehículos, idealmente lo más cerca posible de su lugar de destino. Esto no suele suponer un problema en los suburbios y zonas residenciales de la ciudad, sin embargo en zonas comerciales de alto flujo de personas y vehículos, encontrar un lugar de estacionamiento puede suponer una tarea que requiere un tiempo y esfuerzo no despreciable. Durante la búsqueda de estacionamiento el conductor suele conducir por diferentes calles en forma repetitiva en la espera de que un puesto quede libre para su uso generando un retraso y consumo de recursos el cual podría ser evitado.

Es por esto que el hecho que un conductor pueda conocer el estado de los diferentes puestos de estacionamientos en un sector antes de llegar a él, trae consigo los siguientes beneficios:

- Disminución del tráfico redundante de conductores en búsqueda de puestos de estacionamiento.
- Menor consumo de combustible y en consecuencia un beneficio a nivel económico y medioambiental.
- Mejora en calidad de vida al reducir el tiempo y esfuerzo para estacionar.

Este proyecto de memoria busca generar una solución a esta problemática, haciendo uso de tecnologías de IoT las cuales son descritas en este informe. Los sistemas que aplican tecnologías de la industria 4.0 son también conocidos como sistemas de estacionamientos inteligentes (“Smart Parking” en inglés).

1.3. Objetivos

En base al contexto y problema mencionados anteriormente, este proyecto busca cumplir los siguientes objetivos:

1. **Creación de sistema de monitoreo de estacionamientos:** Implementar un ejemplo de caso de uso de las tecnologías de Internet de las Cosas (IoT) que permita al usuario observar el estado (ocupado o desocupado) de los estacionamientos de un determinado recinto o ubicación.
2. **Integración de servicios de IoT en la nube:** Este sistema integra por lo menos dos servicios de IoT de la nube de Amazon Web Services.
3. **Medir y/o estimar desempeño:** Una vez implementado el sistema realizar medidas y/o estimaciones temporales de desempeño.

En el capítulo 2 se definen requerimientos para el sistema, los cuales sirven de apoyo para comprobar el cumplimiento de los objetivos mencionados.

1.4. Alcance

Para no desviarse del enfoque principal del proyecto, se definen las siguientes afirmaciones acerca de lo que este proyecto busca y lo que no:

Este proyecto es:

- Caso de uso de ejemplo para utilización de servicios de IoT en la nube.
- Estudio del uso, configuración, integración y desempeño de estos servicios.
- Creación de una prueba de concepto del caso de uso a desarrollar.

Este proyecto no es:

- Estudio de métodos de detección de vehículos.
- Sistema de Smart Parking para ser usado en producción.

1.5. Protocolo de comunicación MQTT

El protocolo MQTT (Message Queuing Telemetry Transport) es un protocolo de comunicación ligero y eficiente diseñado para la transmisión de mensajes entre dispositivos con recursos limitados, como sensores y dispositivos IoT.

El protocolo MQTT utiliza un modelo de publicación/suscripción, donde los dispositivos, a través de un agente de mensajes pueden publicar mensajes en un tema (topic) específico y otros dispositivos pueden suscribirse a ese tema para recibir los mensajes correspondientes. Los temas son cadenas de texto que se utilizan para identificar los mensajes y organizar la comunicación. La figura 1.3 muestra un ejemplo de cómo se estructura la comunicación MQTT para transmitir datos de un sensor de temperatura.

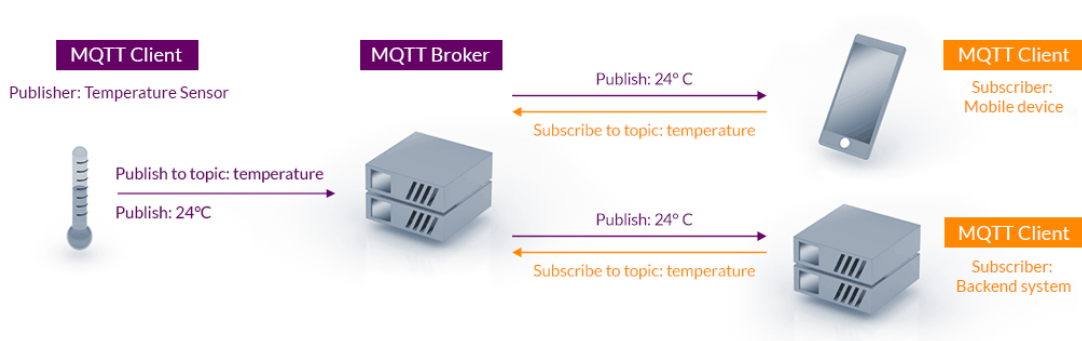


Figura 1.1: Ejemplo de comunicación MQTT

Este protocolo también utiliza un modelo de calidad de servicio (QoS) para garantizar que los mensajes sean entregados de manera confiable y eficiente. El QoS se puede configurar en tres niveles:

- QoS 0: entrega “al menos una vez”, sin confirmación de recepción.
- QoS 1: entrega “exactamente una vez”, con confirmación de recepción.
- QoS 2: entrega “exactamente una vez” y garantiza la entrega ordenada de los mensajes, con confirmación de recepción.

Además, el protocolo MQTT utiliza un enfoque de "mantenimiento de sesión", lo que significa que una vez que se establece una conexión entre el cliente y el servidor, la conexión se mantiene abierta para permitir la transmisión de mensajes de manera continua y eficiente.

Para añadir seguridad a los mensajes MQTT es posible utilizar SSL/TLS en la capa de transporte, ofreciendo un cifrado de extremo a extremo. Esto tiene como ventaja la transferencia segura de la información y como desventaja impacto en el rendimiento al aumentar el costo de procesamiento de los mensajes.

1.6. Proyectos existentes de Smart Parking

En pos de tener una mejor comprensión del estado de arte de las soluciones de "Smart Parking" se presentan a continuación ejemplos desarrollados por otros autores.

1.6.1. Sistema inteligente de estacionamientos desarrollado por Movidex

Movidex, una empresa de desarrollo de Software con oficinas en Estados Unidos e Inglaterra y centros de desarrollo e investigación en Polonia y Ucrania, implementa un sistema de estacionamiento inteligente para su comercialización [10].

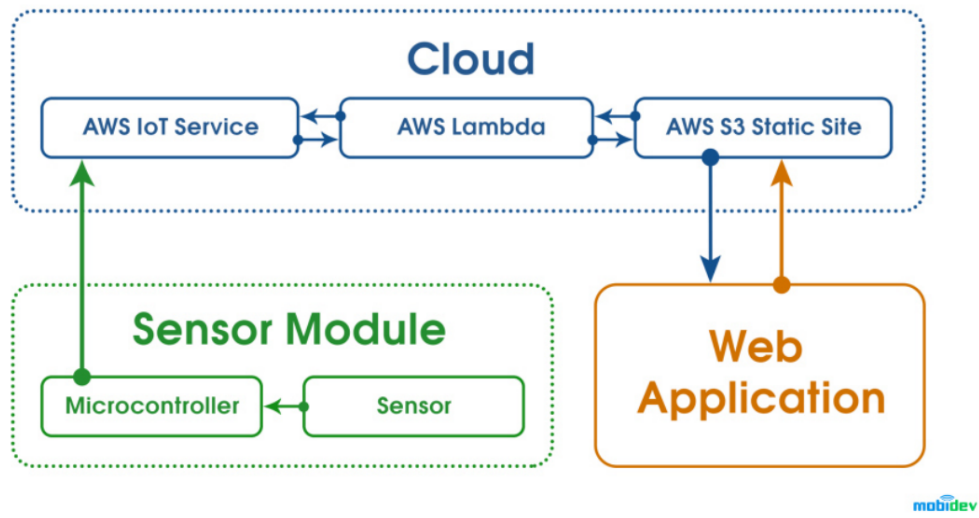


Figura 1.2: Arquitectura del sistema de Smart Parking desarrollado por Mobidev

Los elementos que componen a este ecosistema son los siguientes:

- Sensor: Sensor de ultrasonido HC-SR04 para medición de proximidad [11]
- Microcontrolador: ESP8266 [12]
- Servicios de IoT en la nube: AWS IoT Service [5]
- Servicio de manejo de eventos en la nube: AWS Lambda [13]
- Almacenamiento aplicación web: AWS S3 [14]
- Interfaz de usuario: Aplicación web basada en JavaScript
- Protocolo de red: Message Queue Telemetry Transport (MQTT) [15]

A través del sensor de ultrasonido (HC-SR04) ubicado a nivel de suelo orientado verticalmente hacia arriba se obtiene información sobre el estado de cada puesto de estacionamiento, a través de la detección de la sección posterior o trasera del vehículo. La información de estos sensores es enviada continuamente al servicio AWS IoT por medio del protocolo MQTT (Consultar anexo 1.5). Los estados de cada puesto de estacionamiento están clasificados en tres posibles categorías:

- “Free”(Libre): Si el sensor no detecta ningún objeto a una distancia de 50 cm de él se supone que el puesto de estacionamiento se encuentra libre.
- “Occupied”(Ocupado): Si el sensor detecta la presencia de un objeto en un rango entre 10 cm y 50 cm se asume que el puesto de estacionamiento se encuentra ocupado.

- “Dirty”(Sucio): Si el sensor detecta la presencia de un objeto a una distancia menor a 10 cm se notifica la presencia de un estacionamiento “sucio el cual requiere verificación puesto que puede existir algún objeto que interfiera con el correcto funcionamiento del puesto.

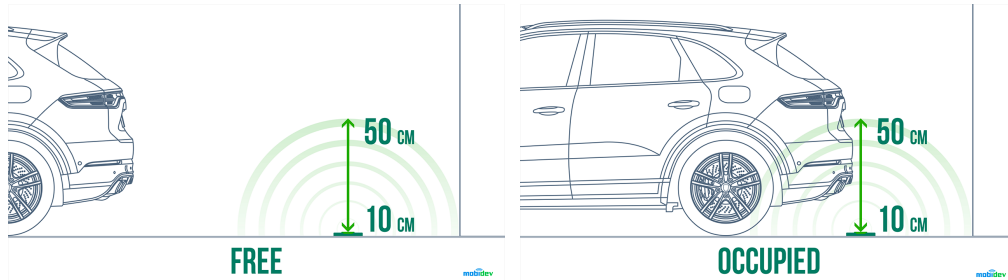


Figura 1.3: Detección del estado de los puestos de estacionamientos

Ya que este proyecto es desarrollado como una prueba de concepto, el artículo de este no aborda detalles, tales como protección de los sensores ante pisadas o neumáticos de los vehículos.

1.6.2. Sistema de Smart Parking desarrollado por Shoban Kumar

Este sistema, cuyo repositorio se encuentra público en GitHub [16], fue publicado en 2016 y su objetivo es proporcionar un sistema de estacionamiento inteligente para gestionar los estacionamientos para centros comerciales, los cuales tienen diferentes edificios de estacionamientos operados por diferentes agentes.

Componentes utilizados en el proyecto:

- Hardware:
 - Raspberry Pi 2 Model B [17]
- Software y Servicios en la Nube:
 - Microsoft Azure [6]
 - Smooch Api [18]
 - Slack [19]
 - Microsoft Windows 10 IoT Core [20]

La figura 1.4 muestra un diagrama de la arquitectura del sistema, el cual integra diferentes servicios de Microsoft Azure, entre ellos Azure IoT Hub que se encarga de

la mensajería entre dispositivos IoT, Azure Active Directory el cual sirve para gestionar grupos de usuarios, entre otros. Además se integran servicios externos como el sistema de mensajería Slack y la API Smooch.

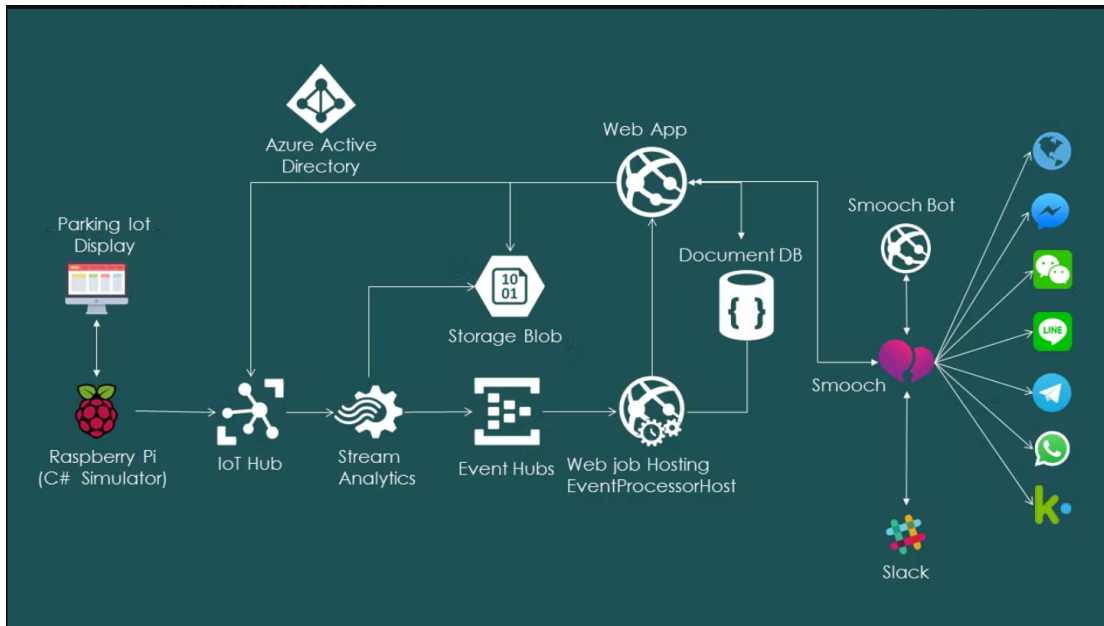


Figura 1.4: Arquitectura del sistema desarrollado por Shoban Kumar

Cada edificio de estacionamientos tiene conectados por cables todos sus sensores a una Raspberry Pi 2 Model B la cual genera la comunicación de los sensores con la nube enviando cada minuto el estado de todos los sensores al Azure IoT Hub. Además por medio de una aplicación UWP (Universal Windows Platform) se muestra en una pantalla conectada a la Raspberry el estado actual de los puestos de estacionamiento, para que el cliente conozca el estado de estos apenas haga ingreso al edificio.

Este proyecto no hace mención alguna de los sensores utilizados, puesto que está diseñado para ser utilizado por cualquier tipo de sensor compatible con la Raspberry. Inclusive este sistema incluye una vista de administrador el cual permite gestionar los sensores utilizados en cada uno de los puestos de estacionamiento.

En la figura 1.5 se pueden observar la vista de la pantalla de la Raspberry y la vista del administrador.

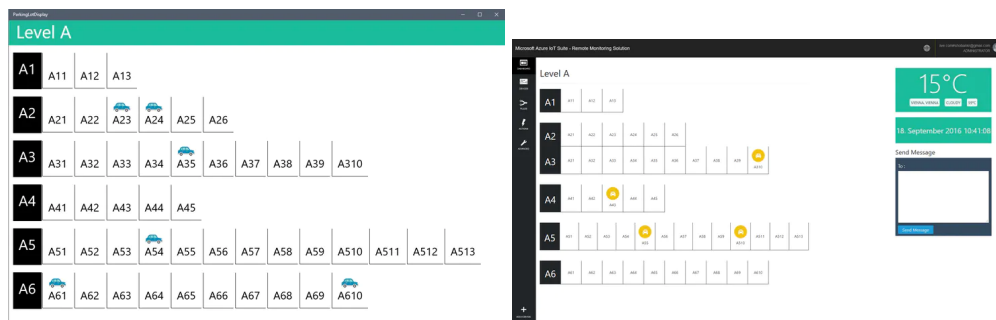


Figura 1.5: Vistas de la pantalla de la Raspberry y de administrador

1.7. Herramientas utilizadas

Para la creación del SMPE se utilizan diferentes herramientas que permitieron su desarrollo. Para el desarrollo de software se utilizan los entornos de desarrollo Visual Studio Code y Android Studio; como línea de comandos se hace uso de la terminal Windows Powershell; para la navegación por los servicios de AWS se utiliza el navegador Google Chrome. Todas estas herramientas son ejecutadas en un laptop “ASUS VivoBook S14” con sistema operativo Windows 10. Para las conexiones realizadas se utiliza una “protoboard” y cables jumper macho-macho y macho-hembra. Además para realizar las pruebas se utiliza un teléfono móvil “Xiaomi Redmi Note 11 Pro 5G” con conexión a Internet por plan de datos.

1.8. Resumen de capítulos siguientes

En los próximos capítulos de este documento se describe en detalle el desarrollo del sistema de estacionamiento inteligente, el cuál se le ha llamado “Sistema de Monitoreo de Puestos de estacionamiento” y mencionado por sus siglas SMPE.

El capítulo número 2 abarca tanto el diseño conceptual del sistema en su totalidad como el de los cuatro subsistemas que lo componen. Para los sistemas y subsistemas se definen los requisitos funcionales y no funcionales de cada uno.

La implementación detallada de cada uno de los subsistemas, los componentes utilizados y su desarrollo paso a paso se encuentra detallado en el capítulo 3.

Para comprobar el correcto funcionamiento de la implementación, en el capítulo

4 se reportan las pruebas para comprobar el correcto funcionamiento del sistema y de sus subsistemas. Además estas se utilizan para determinar el cumplimiento de los requisitos funcionales y no-funcionales. Los resultados de cada una de estas pruebas se encuentran en este mismo capítulo.

Finalmente en el capítulo 5 se encuentra la conclusión de este documento en el cuál se analiza si se cumplen los objetivos de este proyecto y se proponen métodos para escalar el sistema y mejoras a éste que podrían permitirle ser integrado en un entorno de producción.

Posterior a las conclusiones se encuentran los anexos y la bibliografía de este documento.

Capítulo 2

Diseño del SMPE

Para cumplir con los objetivos planteados en la sección 1.3 se definen los requisitos funcionales y no funcionales que deben cumplirse para ello. Los requisitos son planteados para el sistema completo y para cada uno de sus subsistemas (4 en total). Los criterios de aceptación para cada uno de los requerimientos son definidos en el capítulo 4.

2.1. Arquitectura del sistema

En la figura 2.1 se observa un diagrama de contexto, el cual ilustra cómo el sistema interactúa con su entorno. Las entradas del sistema son los cambios de estado de los puestos de estacionamientos a ser monitoreados y la salida es la información del estado de los estacionamientos, la cual es consumida por los usuarios.



Figura 2.1: Diagrama de contexto del SMPE

El diseño interno del SMPE se compone de cuatro subsistemas, los cuales se denominan:

- Subsistema de Sensores
- Subsistema de Dispositivos de Borde
- Subsistema de Nube de IoT
- Subsistema de Aplicación

El diagrama de bloques de la figura 2.2 ilustra la interacción de los subsistemas.

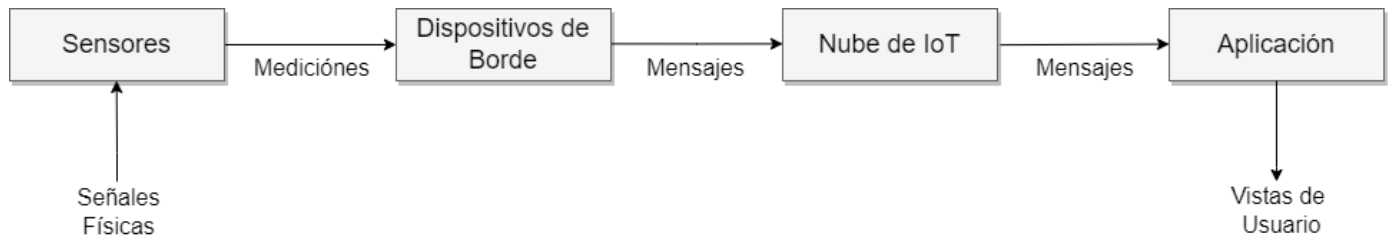


Figura 2.2: Diagrama de bloques de subsistemas del SMPE

A grandes rasgos un conjunto de sensores es utilizado para monitorear un conjunto de puestos de estacionamientos por medio de sus mediciones, las cuales son recibidas e interpretadas por un dispositivo de borde, el cual se comunica con un servicio de nube de IoT por medio de mensajes para notificar cambios en el estado de los estacionamientos. Los mensajes son reenviados por el servicio de nube a una aplicación de usuario, la cual muestra por pantalla el estado de cada uno de los puestos de estacionamiento.

A continuación se explica en mayor detalle el funcionamiento de cada uno de los subsistemas.

2.1.1. Subsistema de Sensores

El Subsistema de Sensores cumple la funcionalidad de detectar la presencia o ausencia de vehículos en los puestos de estacionamiento, por lo tanto compone al conjunto de sensores utilizados para ese fin. Cualquier componente adicional que sea necesaria para el funcionamiento del sensor y enviar sus mediciones al dispositivo de borde conforman igualmente parte de este subsistema. Las entradas son las señales físicas a medirse para detectar el estado de los puestos de estacionamiento y sus salidas la medición realizada a ser procesada por el Subsistema de Dispositivos de Borde.

La comunicación entre el Subsistema de Sensores y el Subsistema de Dispositivos de borde puede realizarse de forma cableada o inalámbrica y los datos se pueden trans-

mitir de forma directa o utilizando un protocolo de comunicación serial. Esto depende del tipo de sensores utilizados para detectar el estado de los puestos de estacionamiento. Para la implementación del SMPE se utilizan sensores de proximidad ultrasonido los cuales se conectan de forma cableada al Subsistema de Dispositivos de borde por medio de sus pins “TRIG” y “ECHO”, esto se explica en mayor detalle en el capítulo 3.1.1.

2.1.2. Subsistema de dispositivos de borde

El Subsistema de Dispositivos de Borde es el encargado de recibir las mediciones del Subsistema de Sensores y procesarlas para determinar de forma cualitativa el estado de los estacionamientos. Si este sistema detecta algún cambio de estado de alguno de los puestos de estacionamiento informa de esto al subsistema de Nube de IoT, a través de un mensaje. Por lo tanto este está compuesto por el conjunto de dispositivos de borde que actúan como intermediarios entre los subsistemas de Sensores y de Nube IoT. Sus entradas son los datos de mediciones del Subsistema de Sensores y sus salidas son los mensajes al Subsistema de Nube de IoT para notificar cambios en los estacionamientos.

En la figura 2.3 se observa un diagrama del funcionamiento general del software que debe ejecutar un dispositivo de borde:

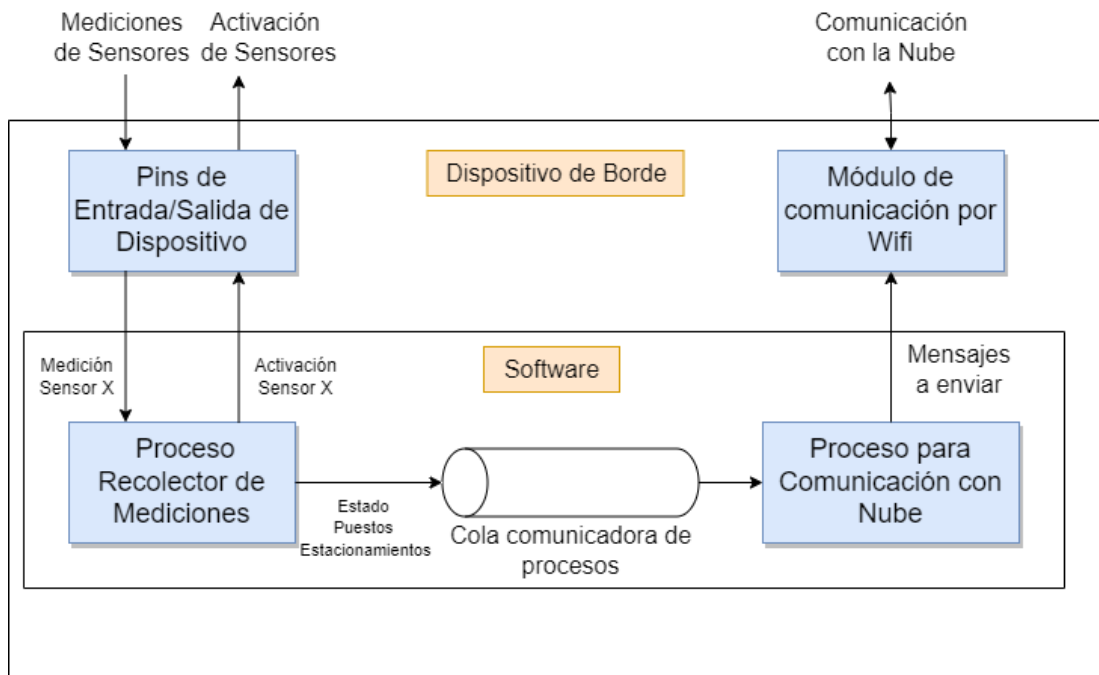


Figura 2.3: Funcionamiento del Software del dispositivo de borde

Como se puede observar, los pins de entada/salida del dispositivo son los encargados de recibir las mediciones y coordinar la activación de los sensores para que realicen mediciones. Estos pins son manejados por el Proceso Recolector de Mediciones, el cual se encarga igualmente de interpretar las mediciones y enviar la información del estado de los puestos de estacionamiento al Proceso para Comunicación con la Nube, el cual se encarga de recibir esta información y enviarla a la Nube por medio del Modulo de comunicación por WiFi.

Ya que ambos, el Proceso Recolector de Mediciones y el Proceso para comunicación con la Nube operan de manera simultánea, los procesos se comunican y se sincronizan a través de una cola.

2.1.3. Subsistema de Nube de IoT

El Subsistema de Nube de IoT es el encargado de comunicar el Subsistema de Dispositivos de Borde con el Subsistema de aplicación, por medio de mensajes de forma remota a través de Internet. Sus entradas son los mensajes recibidos del Subsistema

de Dispositivos de borde y sus salidas son los mensajes con destino al subsistema de aplicación.

Los mensajes son enviados hacen uso del protocolo MQTT como protocolo de mensajería (descripción en capítulo 1.3) y TLS como protocolo de transporte. El contenido de estos es de la forma “puesto:x:y” donde “x” representa el número de estacionamiento el cuál cambió de estado e “y” representa el número del nuevo estado. La numeración de los estados es la siguiente:

- 0: Puesto de estacionamiento libre
- 1: Puesto de estacionamiento ocupado
- 2: No se puede determinar (medición “sucía”)

2.1.4. Subsistema de Aplicación

El Subsistema de Aplicación corresponde a una aplicación que un usuario humano utiliza para observar y monitorear el estado de los puestos de estacionamiento. Sus entradas son los mensajes recibidos del Subsistema de Nube de IoT y sus salidas son las vistas de usuario generadas por la aplicación.

2.2. Requisitos

A continuación se detallan los requisitos funcionales y no funcionales del SMPE:

2.2.1. Requisitos funcionales

1. El sistema identifica el estado de un conjunto de puestos de estacionamientos y los cambios de este.
2. El usuario puede visualizar a través de una aplicación móvil el estado de un conjunto de puestos de estacionamientos simultáneamente.
3. La aplicación móvil muestra el estado de los estacionamientos independiente de la ubicación del usuario, siempre y cuando se posea una conexión activa a Internet.

2.2.2. Requisitos no funcionales

1. El sistema está implementado con al menos dos servicios de IoT de AWS [4].
2. Los cambios de estado de los puestos de estacionamientos son reflejados al usuario en un intervalo de tiempo no mayor a 2 segundos.

2.3. Requisitos de Subsistemas de SMPE

A continuación se detallan los requisitos funcionales y no-funcionales de cada uno de los subsistemas.

2.3.1. Subsistema de Sensores

Requisitos funcionales:

1. El subsistema realiza mediciones físicas que permiten determinar el estado de los puestos de estacionamientos.
2. La medición es entregada al subsistema de dispositivos de borde en un formato compatible con este.

Requisitos no-funcionales: Este subsistema no posee requisitos no-funcionales

2.3.2. Subsistema de Dispositivos de Borde

Requisitos funcionales:

1. Se reciben los datos de medición del Subsistema de Sensores.
2. Los datos son procesados para determinar cualitativamente el estado de los puestos de estacionamiento.
3. Los estados de estacionamiento a reconocer son: *Ocupado*, *Libre* y *Sucio*.
4. El Subsistema envía mensajes al Subsistema de Nube de IoT para notificar cambios en el estado de los estacionamientos.

Requisitos no-funcionales:

1. Los dispositivos de bordes deben tener la capacidad de conectarse a Internet de forma inalámbrica.
2. Los dispositivos de borde a escoger deben estar en la lista de dispositivos validados por AWS [23].

2.3.3. Subsistema de Nube de IoT

Requisitos funcionales

1. Se reciben mensajes del Subsistema de Dispositivos de Borde por medio de un agente de mensajes.
2. Los mensajes recibidos son reenviados al Subsistema de Aplicación.

Requisitos no-funcionales

1. El subsistema debe ser implementado utilizando servicios de la nube de AWS

2.3.4. Subsistema de Aplicación

Requisitos funcionales

1. El subsistema recibe mensajes del Subsistema de Nube de IoT.
2. El sistema reacciona ante los mensajes actualizando la vista del usuario.

Requisitos no-funcionales:

1. La aplicación puede ser usada en cualquier sitio, siempre y cuando exista una conexión a Internet.

Capítulo 3

Implementación

3.1. Componentes utilizados

A continuación se detallan las componentes utilizadas en cada subsistema para cumplir con sus funciones y requerimientos. La figura 3.1 ilustra la implementación de la prueba de concepto del SMPE desarrollado durante el proyecto, el cual tiene la capacidad de monitorear simultáneamente 4 puestos de estacionamiento. Detalles de cómo escalar este sistema a mayor cantidad de puestos de estacionamiento se encuentran en el capítulo 5.

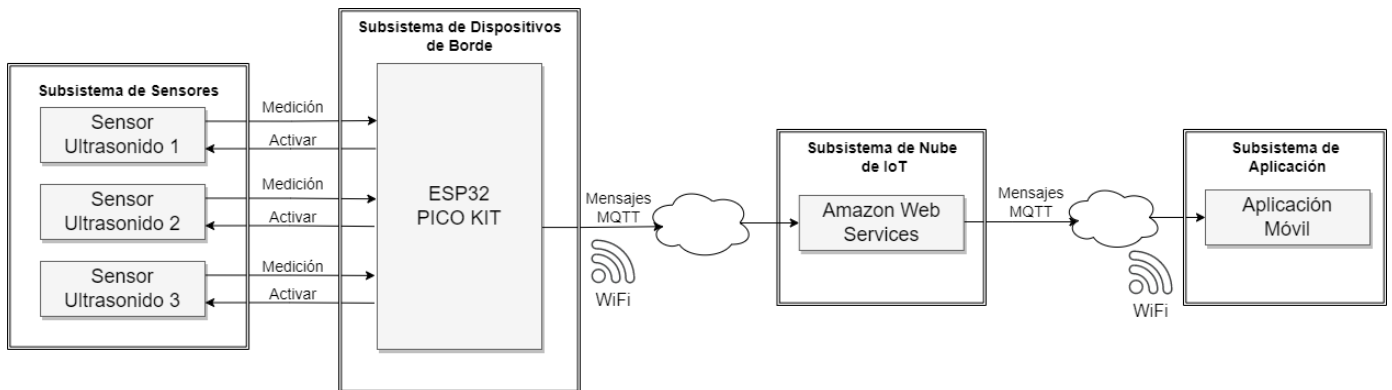


Figura 3.1: Diagrama de bloques de la implementación del SMPE

3.1.1. Subsistema de Sensores

Para detectar la presencia o ausencia de vehículos en sus respectivos puestos de estacionamiento se evalúa la utilización de las siguientes tecnologías:

- Detección de proximidad con sensor óptico.
- Detección de vehículos con magnetómetro.
- Detección de proximidad con sensor ultrasonido.
- Detección de vehículos con cámara de vídeo y visión artificial.

De estas tecnologías se escoge la tercera opción debido a que es una tecnología simple donde sus rangos de medición ofrecen una medición altamente confiable y un tamaño de datos mínimo, además del bajo precio de compra de los sensores. Este tipo de sensor permite la rápida implementación del Subsistema de Sensores para dedicar mayor cantidad de trabajo al Subsistema de Dispositivos de Borde y Nube de IoT, los cuales son el enfoque principal del proyecto.

Los sensores ultrasonido utilizados son sensores HY-SRF05 (observar figura 3.2) [21], los cuales tienen un rango de medición de 2 cm a 450 cm con una resolución de 2 mm. Estos se pueden encontrar en el mercado nacional por 2.489 CLP por unidad (tienda altronics) [24].

Estos sensores operan de la misma forma que el proyecto desarrollado por Movidev [10] (consultar sección 1.6.1), es decir, se utiliza un sensor por puesto de estacionamiento, donde este se instala a nivel de piso orientado hacia arriba (o alternativamente frente al estacionamiento orientado hacia este). Si el sensor detecta un obstáculo a una distancia entre 10 cm y 50 cm se supone que el estacionamiento se encuentra en estado *Ocupado*; en caso que el sensor detecte un obstáculo a más de 50 cm o no detecte ningún obstáculo se supone que el estacionamiento está en estado *Libre*, en caso contrario se supone que el estacionamiento está en estado *Sucio*, es decir, no se puede determinar si el estacionamiento se encuentra libre o ocupado y posiblemente requiera acción correctiva.

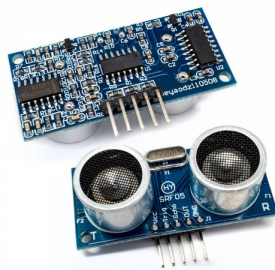


Figura 3.2: Sensor ultrasonido HY-SRF05 [24]

En la figura 3.3 se puede observar la distribución de los pins de los sensores ultrasonido HY-SRF05.

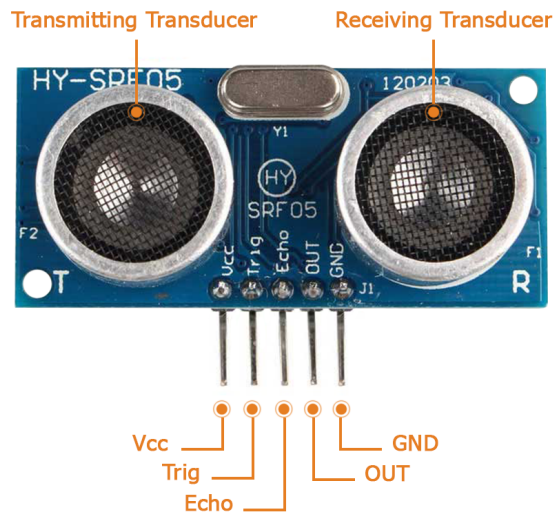


Figura 3.3: Distribución de pins de sensor ultrasonido HY-SRF05

La funcionalidad de cada uno de los pins es la siguiente:

- **VCC:** Conexión a alimentación de 5v continuo.
- **Trig:** Comanda el inicio de la medición si recibe un pulso de mínimo 10 microsegundos.
- **Echo:** Si se detecta un obstáculo genera un pulso de ancho proporcional a la proximidad del sensor a este.
- **OUT:** Genera pulsos eléctricos cuando se envía una onda ultrasónica. Se generan 8 pulsos por medición. (Este pin no es utilizado para el SMPE)
- **GND:** Conexión a tierra

3.1.2. Subsistema de Dispositivos de Borde

Para la selección de Dispositivos de Borde a utilizar se evalúan las siguientes posibilidades:

- STM32L4 Discovery Kit IoT Node [25]
- PIC-IoT Development Board [26]
- SAM E54 Xplained Pro [27]
- Raspberry Pi 3 Model B [28]

- ESP32 PICO Kit [22]

De estos dispositivos se escoge el ESP32 PICO Kit (observar figura 3.4) debido a su alta flexibilidad de configuración, su compatibilidad con los servicios de AWS y su bajo costo de compra (10.927 CLP) [29].

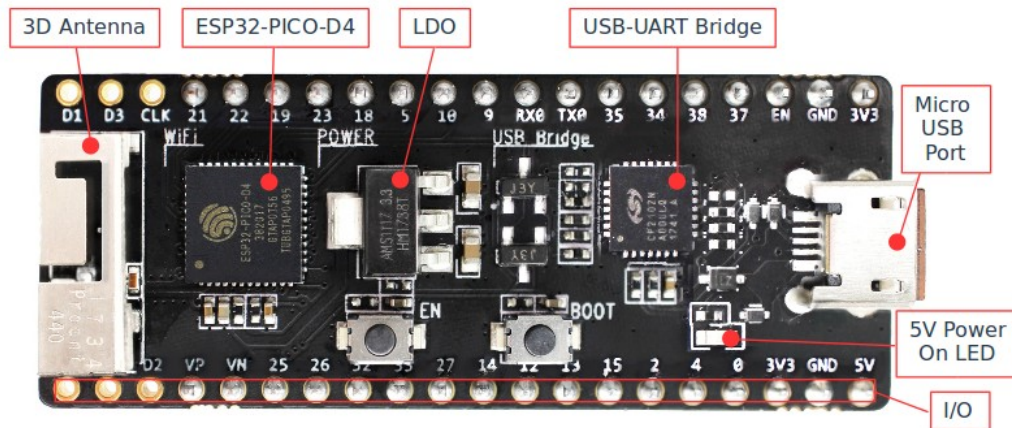


Figura 3.4: Dispositivo ESP32-PICO-KIT V4, [22]

Las principales características de este dispositivo son las siguientes:

- Tamaño de memoria del programa: 4 MB
- Tamaño de RAM de datos: 520 kB
- Voltaje de alimentación operativo: 3.3 a 5V
- Frecuencia: 2.4 a 2.5
- Conectividad por Wifi por medio de antena 3D.
- Interfaz USB con puerto micro USB.
- 28 pins de entrada/salida con periféricos configurables; 7 pins de entrada con periféricos configurables y 5 pins de energía.

La distribución de funcionalidades de los “Pins” del microcontrolador y sus respectivos periféricos se pueden observar en la figura 3.5.

ESP32-PICO-KIT

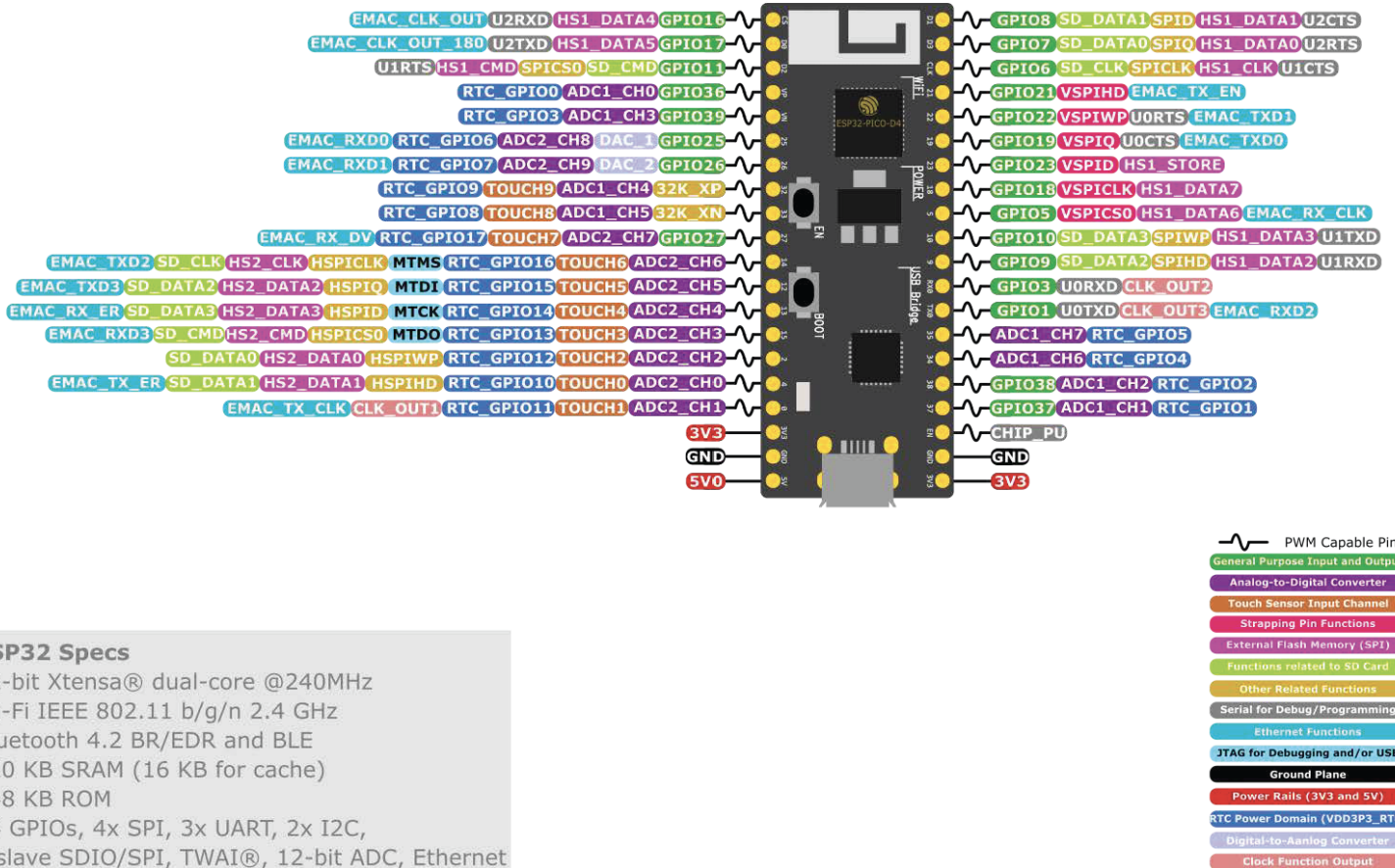


Figura 3.5: Distribución de pines del ESP32-PICO-KIT, [22]

Para facilitar la configuración de este dispositivo y el desarrollo de su software se utiliza el “Espressif IoT Development Framework”(esp-idf) que contiene una API para interactuar con el “Toolchain”, el cual es el encargado de la compilación del código para ESP32. Además este provee herramientas para creación de proyectos, integración de librerías, carga de los archivos binarios al microcontrolador, entre otros [30].

Este “Framework” integra el “Kernel” del sistema operativo a tiempo real FreeRTOS. Este es un sistema operativo para microcontroladores que por medio de sus librerías añade funcionalidades a este como la ejecución de tareas en paralelo (“Task”) y medios de comunicación para ellas como colas (“Queues”). [31]. Estas funcionalidades se encuentran presentes en el código fuente del dispositivo de borde.

3.1.3. Subsistema de Nube de IoT

Para el subsistema de nube de IoT se utilizan los siguientes servicios de AWS.

- **IoT Core:** Este servicio provee un agente de mensajes para que dispositivos IoT intercambien mensajes MQTT con este [5]. Los dispositivos pueden publicar mensajes o suscribirse a estos por medio de un tópico (consultar anexo 1.5). En el contexto de este proyecto los dispositivos son el microcontrolador (publica mensajes) y el teléfono móvil en el que se utiliza la aplicación móvil (se suscribe a mensajes).
- **IoT Device Management:** Servicio encargado de administrar los dispositivos IoT, a través de la creación de “Objetos” los cuales pueden ser físicos o lógicos [32]. Estos objetos pueden crearse individualmente o flotas de ellos, a la vez estos pueden ser organizados por grupos y etiquetas. Esto facilita al administrador del sistema la gestión de grandes flotas de dispositivos de forma sistemática y ordenada.

Para que un dispositivo pueda publicar o suscribirse a mensajes debe poseer un certificado el cual está asociado a una política la cual define qué acciones puede realizar. Para efectos de este proyecto los dispositivos de borde deben estar autorizados a conectarse a “AWS IoT Core” [5]. Esta configuración es realizada utilizando el servicio “IoT Device Management”. El paso a paso de esta configuración se encuentra en el capítulo 3.2.4.

La aplicación se conecta a AWS por medio de un “WebSocket” y la autenticación a este se realiza a través del servicio “Amazon Cognito” [33], el cual es un servicio para gestionar la identidad y el acceso de los clientes por medio del uso de roles. Esto permite a los usuarios de aplicaciones que utilizan servicios de AWS conectarse a esta sin la necesidad de utilizar credenciales. La configuración de los roles se observa en el capítulo 3.2.4.

3.1.4. Subsistema de Aplicación

Para el subsistema de aplicación se desarrolla una aplicación nativa para dispositivos Android, la cual se conecta y recibe mensajes del servicio “AWS IoT Core” [5].

para mostrar por pantalla el estado actualizado de los puestos de estacionamiento.

Para el desarrollo de esta se utiliza la herramienta “Android Studio” [34], el entorno oficial para desarrollo de aplicaciones en este sistema operativo. Como base para el código fuente se utiliza el proyecto de ejemplo del repositorio *aws-sdk-android-samples* del usuario *felipemeriga* del sitio “Github” [35]. Este proyecto es adaptado a los requerimientos del Subsistema de Aplicación.

3.2. Desarrollo de los submódulos

3.2.1. Conexión de los sensores

Para conectar los sensores al microcontrolador ESP32-PICO se realizan las conexiones en base al diagrama de conexiones de la figura 3.6, este diagrama asume que el puerto “USB-Micro” del microcontrolador se encuentra conectado a una fuente de poder.

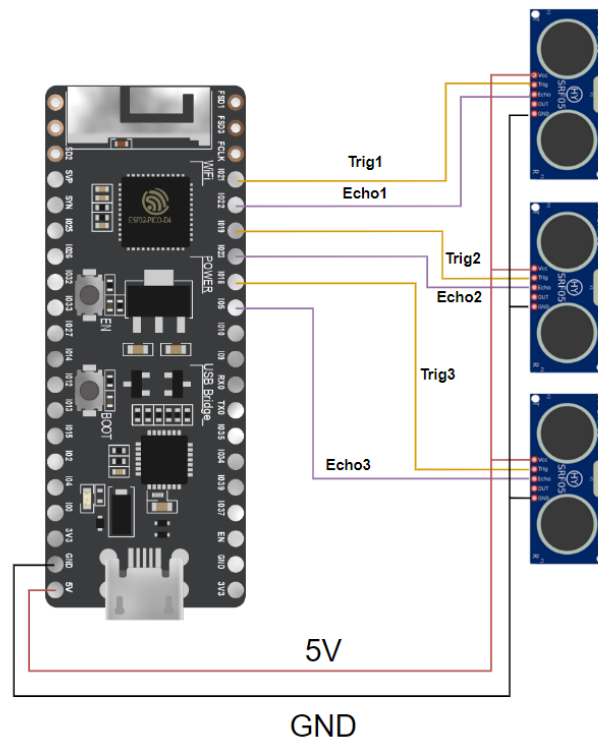


Figura 3.6: Diagrama de conexiones de los sensores ultrasonido al microcontrolador

La figura 3.7 muestra una fotografía del conjunto de sensores conectados al Dispositivo de Borde:

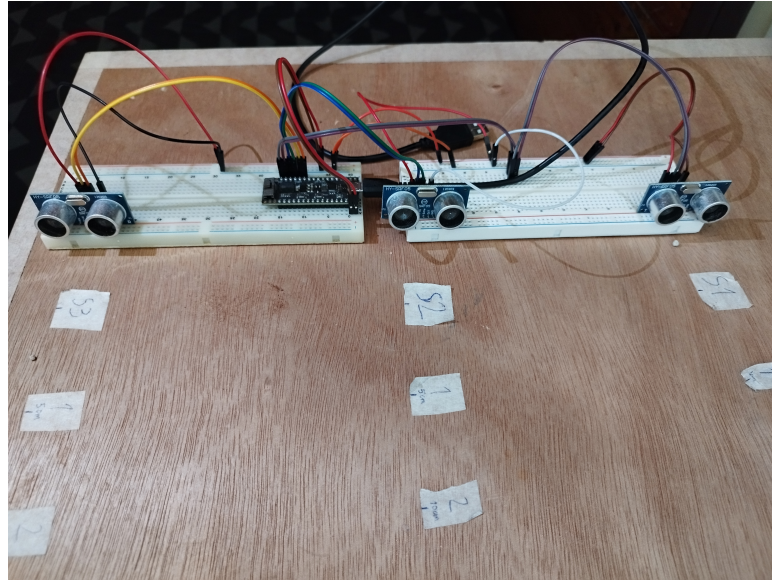


Figura 3.7: Fotografía de conjunto de sensores conectados al Dispositivo de Borde

3.2.2. Instalación y configuración de esp-idf

Para la instalación y configuración del “framework” esp-idf se utiliza como guía los pasos sugeridos en la documentación oficial de AWS [36] los cuales se detallan a continuación. Estos pasos se realizan en un computador “ASUS VivoBook S14” con el sistema operativo “Windows 10” y se supone que el sistema tiene instalado Git y algún compilador de C con sus variables de entorno ya configuradas.

Como primer paso se crea una carpeta en el sistema para almacenar los diferentes archivos del proyecto y dentro de ella se clona el repositorio [37] de Amazon-FreeRTOS haciendo uso del siguiente comando:

```
git clone --recursive https://github.com/aws/amazon-freertos.git
```

Al ejecutarse exitosamente este comando crea la carpeta *amazon-freertos* que contiene los archivos y directorios del repositorio. Posterior a esto se abre el instalador de esp-idf [38] y se realizan los siguientes pasos

1. Avanzar la instalación hasta la opción “Descarga o usa ESP-IDF”.

2. Una vez en esta opción, seleccionar “Usar un directorio existente de ESP-IDF” y seleccionar el directorio *<Directorio de trabajo>\amazon-freertos\vendors\espressif\esp-idf*. Esto instalará el “framework” utilizando el kernel de FreeRTOS de amazon, en vez de el kernel de Espressif.
3. Luego escoger la carpeta de destino de la instalación en un directorio diferente a *amazon-freertos* o sus sub-directorios. Los archivos son instalados en el destino en un directorio llamado *Espressif*.
4. Avanzar con la instalación utilizando valores por defecto.
5. Finalizar instalación.

Una vez finalizada la instalación es posible ejecutar el ambiente ESP-IDF Powershell, el cual consiste en una instancia de Powershell que al iniciar configura las variables de entorno necesarias para utilizar ESP-IDF. La figura 3.8 muestra un ejemplo de ejecución del ESP-IDF Powershell. Este permite utilizar los siguientes comandos en directorios que contengan un archivo CMakeLists [39]:

- *idf.py build*: Construye y compila el proyecto, las salidas generadas se guardan en el sub-directorio *build*.
- *idf.py flash*: Carga el programa al microcontrolador, a través de su puerto serial.
- *idf.py monitor*: Monitorea el puerto serial del microcontrolador.
- *idf.py build flash monitor*: Ejecuta los comandos anteriores uno tras otro.

```
ESP-IDF 4.4 PowerShell
git version 2.34.1.windows.1
Setting IDF_PATH: C:\Espressif\frameworks\esp-idf-v4.4.3-2
Adding ESP-IDF tools to PATH...

Name                                Value
----                                -
OPENOCD_SCRIPTS                     C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\openocd-esp32\v0.11.0-esp32-...
IDF_CCACHE_ENABLE                   1
IDF_PYTHON_ENV_PATH                 C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\python_env\idf4.4_py3.8_env

Added to PATH
-----
C:\Espressif\frameworks\esp-idf-v4.4.3-2\components\esptool_py\esptool
C:\Espressif\frameworks\esp-idf-v4.4.3-2\components\app_update
C:\Espressif\frameworks\esp-idf-v4.4.3-2\components\espcoredump
C:\Espressif\frameworks\esp-idf-v4.4.3-2\components\partition_table
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\xtensa-esp32-elf\esp-2021r2-patch5-8.4.0\xtensa-esp32-elf\bin
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\xtensa-esp32s2-elf\esp-2021r2-patch5-8.4.0\xtensa-esp32s2-elf\bin
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\xtensa-esp32s3-elf\esp-2021r2-patch5-8.4.0\xtensa-esp32s3-elf\bin
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\riscv32-esp-elf\esp-2021r2-patch5-8.4.0\riscv32-esp-elf\bin
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\esp32ulp-elf\2.35_20220830\esp32ulp-elf\bin
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\cmake\3.23.1\bin
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\openocd-esp32\v0.11.0-esp32-20220706\openocd-esp32\bin
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\ninja\1.10.2\
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\idf-exe\1.0.3\
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\ccache\4.3\ccache-4.3-windows-64
C:\Users\Esteban\Desktop\ArchivosMemoria\Espresif\tools\dfu-util\0.9\dfu-util-0.9-win64
C:\Espressif\frameworks\esp-idf-v4.4.3-2\tools
Checking if Python packages are up to date...
Python requirements from C:\Espressif\frameworks\esp-idf-v4.4.3-2\requirements.txt are satisfied.

Done! You can now compile ESP-IDF projects.
Go to the project directory and run:
    idf.py build

PS C:\Espressif\frameworks\esp-idf-v4.4.3-2>
```

Figura 3.8: Ejemplo de ejecución del ambiente ESP-IDF Powershell

Para este ejemplo se crea una nueva carpeta llamada "SmartParking", en la cual se clona el repositorio de amazon-freertos y se instala el "framework" ESP-IDF.

Luego de instalar exitosamente el ESP-IDF se crea un nuevo proyecto llamado "smart_parking" con su código fuente principal que se encuentra en el archivo "smart_parking.c". Para crear este proyecto se copia el directorio del proyecto de ejemplo en la ruta <Directorio de trabajo>\amazon-freertos \vendors \espressif \esp-idf \examples \get-started \hello_world y se renombran y adaptan sus respectivos archivos.

3.2.3. Script desarrollado para microcontrolador

El archivo *smart_parking.c* que contiene el código fuente principal del Sistema de Dispositivos de Borde, cumple las funcionalidades descritas en la sección 2.3.2. Éste código es construido a partir del proyecto de ejemplo llamado “subscribe-publish” desarrollado por Espressif el cual es accesible públicamente a través del repositorio Github “espressif/esp-aws-iot” [40]. El código fuente desarrollado puede ser observado en el Anexo A.

A grandes rasgos el código fuente de este archivo en su función principal *app_main* crea dos tareas, una llamada *ultrasonic_task* encargada de realizar las mediciones de los sensores de ultrasonido y otra llamada *aws_iot_task* encargada de mantener la comunicación con la nube de AWS y enviar mensajes a este cuando los sensores detecten un cambio de estado en los puestos de estacionamiento. Estas dos tareas se comunican y sincronizan entre ellas por medio de una cola (ver figura 2.3). La capacidad de este código de ejecutar tareas en paralelo utilizando tareas y comunicar procesos a través de colas, es posible haciendo uso del sistema operativo a tiempo real “FreeRTOS” [31].

A continuación se describe la funcionalidad de los diferentes fragmentos de código de este archivo:

Función *app_main*: Esta función es la principal del código, la primera en ejecutarse del código fuente luego de las definiciones globales y los macros (observar figura 3.9). Esta realiza las siguientes tareas:

- Se llama a la función *xQueueCreate* [41], la cual crea una cola que comunica a las tareas *ultrasonic_task* con *aws_iot_task*. Esta función acepta dos parámetros, el primero es la cantidad máxima de elementos que la cola puede almacenar a la vez y el segundo es el tamaño necesario, en bytes, para almacenar un elemento en la cola. Para estos parámetros se utilizan los valores 10 y *sizeof(Sensor_info)* respectivamente. *Sensor_info* corresponde a una estructura usada para monitorear el estado de los sensores (Más detalles en la descripción de *ultrasonic_task*).
- Si ocurre un error en la creación de la cola se aborta el programa y se reinicia el microcontrolador.
- Se llama a la función *initialise_wifi*, encargada de configurar al microcontrolador para conectarse a la red wifi con las credenciales definidas en *WIFI_SSID*

y *WIFI_PASS*, también registra eventos los cuales son manejados por la función *wifi_event_handler*.

- Se utiliza la función dos veces *xTaskCreate* [42] para crear las tareas *ultrasonic_task* y *aws_iot_task*, esta función tiene el siguiente prototipo:

```
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        configSTACK_DEPTH_TYPE usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask  
                        );
```

Donde sus parámetros tienen la siguiente funcionalidad:

1. *pvTaskCode*: Puntero a la función que maneja la tarea.
2. *pcName*: Nombre de la tarea a ser creada.
3. *usStackDepth*: Número de palabras (palabras de máquina) a asignar al “stack” de la tarea.
4. *pvParameters*: Parámetros a ser pasados a la tarea.
5. *uxPriority*: Prioridad de la tarea donde un mayor número denota a una tarea con mayor prioridad, siendo 0 la prioridad más baja.
6. *pxCreatedTask*: Parámetro opcional para ser utilizado para pasar la función encargada de la tarea como parámetro a otras funciones.

Para la tarea *aws_iot_task* se asignan 8192 palabras para su “stack” y prioridad 10 y para la tarea *ultrasonic_task* se asignan 2048 palabras a su “stack” y prioridad 2. La cantidad de palabras utilizadas en cada tareas fueron determinadas de forma empírica.

```

void app_main()
{
    //Creación de cola que comunica tareas aws_iot_task y ultrasonic_task
    aws_iot_evt_queue = xQueueCreate(10, sizeof(Sensors_info));

    // Si cola no es creada exitosamente abortar
    if(aws_iot_evt_queue == NULL){
        ESP_LOGE(TAG, "Error creando las colas\n");
        abort();
    }

    // Inicializar NVS.
    esp_err_t err = nvs_flash_init();
    if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        err = nvs_flash_init();
    }
    ESP_ERROR_CHECK( err );

    //Inicializar wifi
    initialise_wifi();

    //Crear tarea encargada de mantener la conexión con AWS y enviar mensajes a este
    xTaskCreate(aws_iot_task, "aws_iot_task", 8192, NULL, 10, NULL);

    //Crear tarea encargada de leer datos de los sensores e interpretarlos
    xTaskCreate(ultrasonic_task, "ultrasonic_task", 1024*2, NULL, 2, NULL);
}

```

Figura 3.9: Código de función *app_main*

Función *ultrasonic task*: Esta función es la encargada de manejar la tarea “ultrasonic task” (la tarea y la función pueden tener nombres diferentes, en este caso son iguales), esta tarea es ejecutada en paralelo a la tarea *aws_iot_task*. Esta realiza los siguientes pasos:

Algoritmo 1 ultrasonic_task

```
1: Inicializar los relojes de inicio y fin.
2: Esperar 10 segundos para permitir que la tarea aws_iot_task comience.
3: Crear un array de sensores para cada lugar de estacionamiento.
4: Crear un array de distancias medidas por cada sensor.
5: Crear una estructura Sensors_info para monitorear los sensores.
6: for cada lugar de estacionamiento do
7:   Inicializar el sensor y sus pines “trigger” y “echo”.
8:   Agregar el sensor al arreglo de sensores.
9:   if hay un error al inicializar el sensor then
10:    Imprimir un mensaje de error.
11:   end if
12:   Inicializar el arreglo de distancia para el sensor.
13:   Inicializar la estructura sensors_info para el sensor.
14: end for
15: loop
16:   for cada sensor en el arreglo de sensores do
17:     Obtener el estado del lugar de estacionamiento y el estado del sensor.
18:     Medir la distancia usando el sensor.
19:     if hay un error al medir la distancia then
20:       Establecer el estado del lugar de estacionamiento como SUCIO o LIBRE,
       dependiendo del error.
21:       if el error no es por tiempo de espera agotado (“timeout”) then
22:         Imprimir el mensaje de error
23:       end if
24:     else
25:       Determinar el estado del lugar de estacionamiento en función de la distancia
       medida.
26:       Imprimir el número de sensor, estado del sensor, estado del lugar de esta-
       cionamiento y distancia.
27:     end if
28:     if el estado del lugar de estacionamiento ha cambiado then
29:       Actualizar la estructura sensors_info para el sensor.
30:       Enviar la estructura sensors_info a la aws_iot_evt_queue.
31:       Imprimir un mensaje de registro indicando que el mensaje ha sido enviado.
32:     end if
33:     Esperar 100ms.
34:   end for
35: end loop
```

Función *aws_iot_task*: Esta función es la encargada de mantener activa la conexión con AWS y publicar mensajes a este, realizando los siguientes pasos:

Algoritmo 2 aws_iot_task

```
1: Crear cliente de AWS IoT.
2: Configuración de parámetros de cliente y conexión MQTT.
3: Esperar a que dispositivo se encuentre conectado a Wifi.
4: Conectar dispositivo a AWS.
5: Se define el tópico de los mensajes como “esp32-1”.
6: while No hay error en envío de mensajes do
7:   Verifica el estado de la conexión con aws por un máximo de 100ms.
8:   if Cliente intenta reconectarse then
9:     Saltar a la siguiente iteración.
10:  end if
11:  if Si la conexión fue exitosa then
12:    Acceder a la cola aws_iot_queue.
13:    if Se recibe un mensaje de la cola then
14:      Guardar mensaje en estructura sensors_info
15:      Preparar carga del mensaje
16:      Publicar mensaje a AWS.
17:      if Mensaje es publicado exitosamente then
18:        Marcar sensor como actualizado,
19:        Continuar iterando.
20:      else if Hay un error en publicar el mensaje then
21:        if Error de tiempo de espera (timeout) then
22:          Reintentar publicar el mensaje hasta 3 veces
23:          if El mensaje es enviado en alguno de los 3 intentos then
24:            Marcar sensor como actualizado,
25:            Continuar iterando.
26:          else
27:            Abortar y reiniciar dispositivo.
28:          end if
29:        else
30:          Mostrar error o advertencia dependiendo el error. En caso de ser error
          abortar.
31:        end if
32:      else
33:        Mostrar error o advertencia dependiendo el error. En caso de ser error
        abortar.
34:      end if
35:    end if
36:  end if
37:  if El cliente se reconecta a la red then
38:    Revisar estado de cada uno de los estacionamientos y de ser necesario actua-
    lizar información y notificar a la nube.
39:  end if
40: end while
```

3.2.4. Configuración de nube de AWS

Para configurar la nube de AWS y cumplir con los requerimientos del Subsistema de Nube de IoT se realizan los siguiente pasos:

- Crear una cuenta de AWS siguiendo los pasos de la documentación oficial [55].
- Se Ingresa a la cuenta de AWS utilizando las credenciales del usuario raíz.
- Se Ingresa “IoT Device Management” en la barra de búsqueda y en la sección Servicios, se elige el servicio con el mismo nombre.
- Esto despliega una lista de los objetos IoT creados por este servicio. Luego se hace click en la opción “Crear objetos”.
- En la vista que se despliega se selecciona la opción “Crear un único objeto” del campo “Número de objetos que crear”.
- En la siguiente sección se rellena el campo “Nombre del objeto” como “ESP32_for_SmartParking_1”, todas las otras opciones son dejadas por defecto y se pulsa el botón “siguiente”.
- Como siguiente paso se escoge la opción por defecto “Generar automáticamente un certificado nuevo (recomendado)” y se pulsa el botón “Siguiente”.
- A continuación se pide al usuario asociar una política al certificado a crearse. Para este objeto se crea una nueva política, por lo cual se pulsa el botón “Crear Política” la cual abre una nueva pestaña en el navegador para la creación de esta. La nueva política es creada con el nombre “ESP32_for_SmartParking_Policy”. Los efectos de la política pueden configurarse utilizando el creador o escribiendo el documento de la política en formato JSON (observar figuras 3.10 y 3.11). Esta es configurada para autorizar al cliente “ESP32_for_SmartParking_1” conectarse a AWS a través del punto de enlace especificado. A través de este mismo autoriza a este cliente a publicar mensajes y suscribirse a estos utilizando el tópico “esp32/1”. Cualquier otro cliente que intente conectarse a este punto de enlace será rechazado y el cliente autorizado solo puede comunicarse con AWS por medio del autorizado, esto permite al sistema funcionar de manera más segura.
- Luego de configurar la política y pulsar el botón “Crear” ya es posible asignar esta al objeto a crearse y pulsar “Crear un objeto”.

- Crear política

Información

Las políticas de AWS IoT Core permiten administrar el acceso a las operaciones del plano de datos de AWS IoT Core.

Propiedades de la política

AWS IoT Core admite políticas con nombre para que muchas identidades puedan hacer referencia al mismo documento de políticas.

Nombre de la política

ESP32_for_SmartParking_Policy

Un nombre de política es una cadena alfanumérica que también puede contener caracteres de punto (.), coma (,), guion (-), guion bajo (_), signo más (+), signo igual (=) y arroba (@), pero sin espacios.

▼ Etiquetas: opcional

No hay etiquetas asociadas al recurso.

Agregar nueva etiqueta

Puede agregar hasta 50 etiquetas más

Instrucciones de política

Ejemplos de políticas

Documento de política

Información

Una política de AWS IoT contiene una o varias instrucciones de política. Cada instrucción de política contiene acciones, recursos y un efecto que concede o deniega las acciones por parte de los recursos.

Efecto de la política

Allow

▼

Acción de la política

iot:Connect

Recurso de la política

arn:aws:iot:us-east-1:396713480151:client/ESP32_for_SmartParking_1

Eliminar

Efecto de la política

Allow

▼

Acción de la política

iot:Subscribe

Recurso de la política

arn:aws:iot:us-east-1:396713480151:topicfilter/esp32/1

Eliminar

Efecto de la política

Allow

▼

Acción de la política

iot:Receive

Recurso de la política

arn:aws:iot:us-east-1:396713480151:topic/esp32/1

Eliminar

Agregar nueva instrucción

Cancelar

Crear

34

A continuación se muestran los pasos necesarios para la configuración de la nube de AWS para satisfacer los requerimientos del Subsistema de aplicación.

- Entrar a AWS, buscar en la barra de búsqueda “Amazon Cognito” y entrar a este servicio.
- Una vez dentro de este servicio pulsar el botón ‘Administrar grupos de identidades’ y en la página resultante pulsar el botón “Crear nuevo grupo de identidades.
- En el formulario de creación de grupo de identidades dar un nombre a esta y asegurarse que la opción “Habilitar el acceso a identidades sin autenticar” se encuentre activada, esto permite a conceder credenciales a AWS a cualquier persona con acceso a Internet, esto permite a los usuarios poder utilizar la aplicación sin necesidad de crear credenciales. El resto de las opciones son dejadas por defecto y se pulsa el botón “Crear un grupo”. Para efectos de este proyecto se nombre al grupo de usuarios como “SmartParkingAndroidApp”.
- En la página resultante se puede observar y editar los dos roles que serán creados en el servicio de administración de roles y acceso (Identity and Access Management o IAM) desplegando la opción “Ver los detalles”. En este paso se utilizan los nombres sugeridos por defecto, es decir, “Cognito_SmartParkingAndroidAppAuth_Role” y “Cognito_SmartParkingAndroidAppUnauth_Role”. En la figura 3.13 se observan estos roles creados exitosamente.
- En la siguiente página copiar y guardar en un lugar seguro el “ID del grupo de identidades”, este es necesario en el código fuente de la aplicación y se encuentra en la sección “Obtención de credenciales de AWS” de la página resultante.
- Luego debe asociarse una política al rol no autenticado, es decir, al rol llamado “Cognito_SmartParkingAndroidAppUnauth_Role”. Para realizar esto, se accede al servicio IAM y se pulsa el hipervínculo del rol. En la página que se despliega en la sección “Permisos” pulsar el desplegable “Añadir Permisos” y luego de las opciones que se despliegan seleccionar “Asociar políticas”.
- Este último paso desplegará una lista llamada “Otras políticas de permisos”, en esta buscar la política llamada “AWSIoTFullAccess”. Seleccionar esta y pulsar el botón “Asociar políticas”. Para un sistema en producción es importante seleccionar una política sólo con permisos de lecturas, esta política es seleccionada solamente para simplificar el proceso de desarrollo. En la figura 3.14 se puede observar la política asociada exitosamente.

El código fuente de la aplicación de usuario posee dos archivos principales, el primero es “activity_main.xml” el cuál describe los elementos visuales de la vista de la aplicación y su disposición. Este archivo es adaptado haciendo uso de la interfaz gráfica de “Android Studio”, el cual automáticamente actualiza el archivo con extension “.xml”. En la figura 3.15 se observa la interfaz gráfica adaptada desde el entorno de desarrollo.

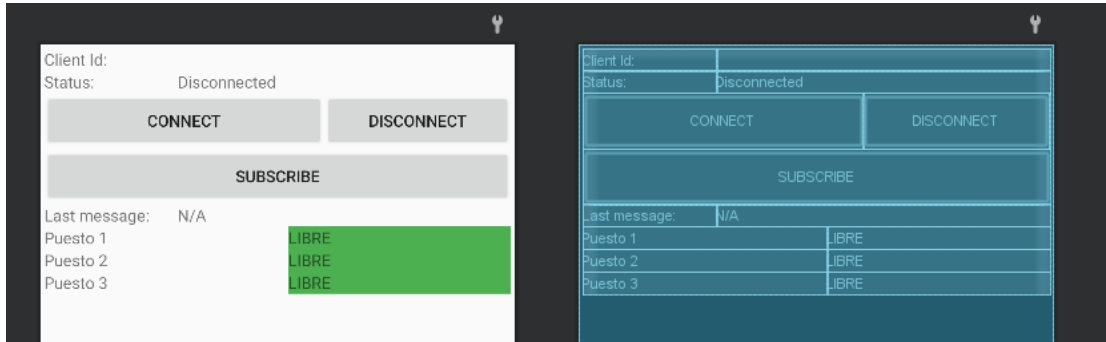


Figura 3.15: Diseño de interfaz gráfica de aplicación de usuario, observado desde “Android Studio”

El otro archivo principal es “PubSubActivity.java”, el cual contiene el código fuente de la lógica de la aplicación encapsulada en la clase *PubSubActivity*. La funcionalidad de este código se puede observar en el diagrama de flujo de la figura 3.16. Información más detallada de los atributos y métodos de la clase definida en este archivo se pueden observar en el anexo C.

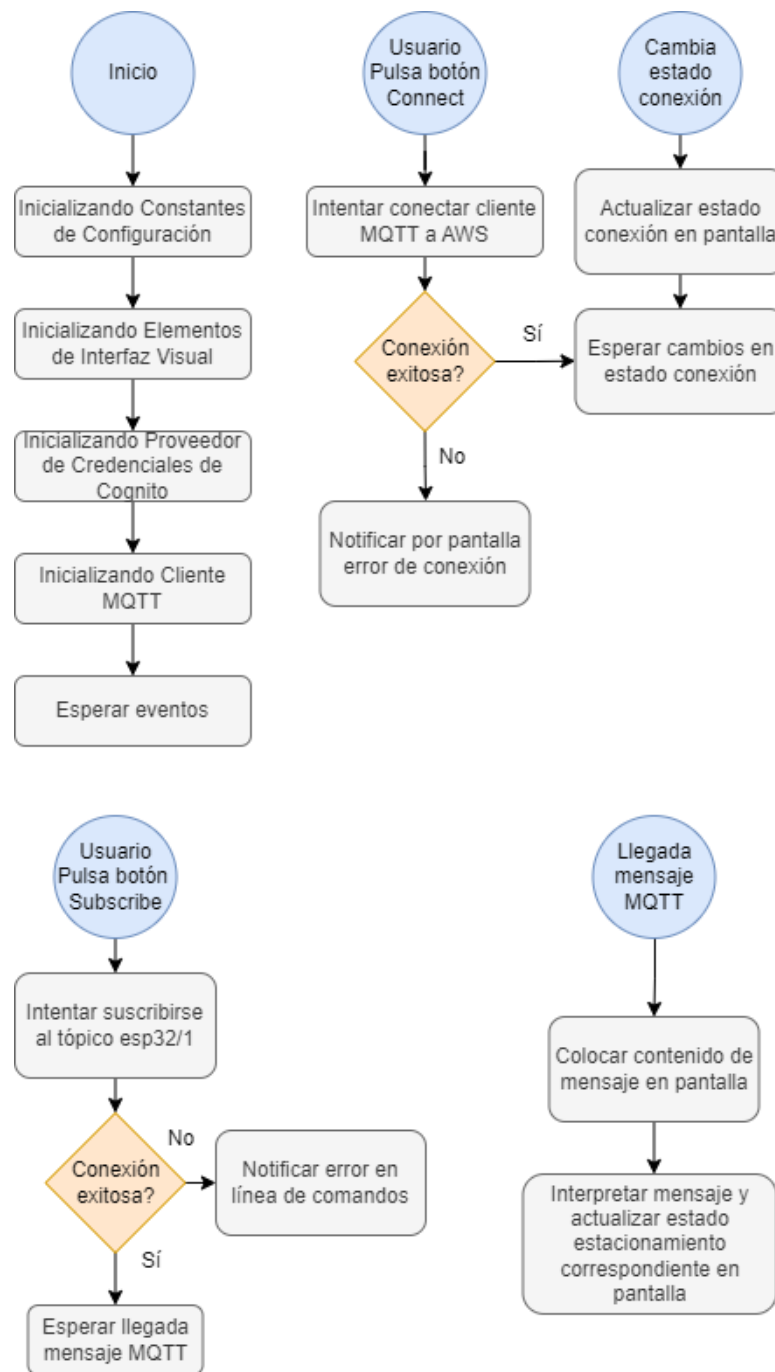


Figura 3.16: Diagrama de flujo de código del archivo PubSubActivity.java

En este diagrama de flujo los círculos representan los diferentes eventos que pueden ocurrir durante la ejecución del programa.

Capítulo 4

Pruebas y resultados

Una vez implementado el sistema se realizan pruebas para comprobar el cumplimiento de los requerimientos funcionales y no funcionales de cada uno de los subsistemas y del sistema en su totalidad.

4.1. Funcionamiento de los sensores

Para cumplir con los requisitos funcionales 1 y 2 del Subsistema de Sensores y los requisitos funcionales 1, 2, 3 y 4 del Subsistema de Dispositivos de Borde, definidos en la sección 2.3.1 y 2.3.2 respectivamente, se utilizan los siguientes criterios de aceptación.

1. Todos los sensores ultrasonido realizan mediciones de proximidad en el rango [5 cm y 100 cm] con un error porcentual promedio (considerando mediciones en intervalos de 5 cm) no mayor al 5%.
2. Si se conectan los sensores a un microcontrolador conectado por USB a una computadora, se pueden observar las mediciones de los sensores en la línea de comandos.

Por lo tanto la prueba a realizar para comprobar estos criterios se realiza de la siguiente manera:

- Comentar las líneas del archivo *smart_parking.c* que contiene la creación de la tarea *aws_iot_task* (llamado a función *xTaskCreate*) y el llamado a la función *xQueueSend* de la tarea *ultrasonic_task* (comentar toda la sentencia “if” que lo

contiene). Esto causará que éste código ejecute solamente la tarea *ultrasonic_task* sin realizar la comunicación con AWS permitiendo probar las mediciones de los sensores y el procesamiento de estas realizada por el microcontrolador.

- Realizar las conexiones ilustradas en la figura 3.6.
- Preparar un entorno libre de obstáculos el cuál en una superficie plana (puede ser en el suelo, un escritorio o una mesa por ejemplo) contenga 3 filas de 20 marcas dispuestas en línea recta distanciadas entre ellas a 5 cm. Cada línea es utilizada por un sensor diferente.
- Ejecutar el proyecto en una línea de comandos con sus variables de entorno configuradas (más información en la sección 3.2.2) ubicándose en el directorio del proyecto y haciendo uso del comando:

```
idf.py build flash monitor
```

- Una vez en ejecución orientar cada uno de los sensores a su respectiva área demarcada, colocar un obstáculo en la primera marca de la primera fila y anotar la distancia medida por el sensor y el estado del “estacionamiento” notificados por el microcontrolador a través de la terminal. Este paso se repite en cada una de las marcas para cada uno de los sensores.
- Finalmente con los valores de medición, hacer una tabla y obtener el error porcentual promedio para cada sensor y asegurar que los estados notificados por el microcontrolador son correctos.

La figura 4.1 muestra el área demarcada para realizar los pasos mencionados junto al circuito armado y el obstáculo que se utiliza para realizar las mediciones. Los resultados obtenidos de las mediciones se pueden observar en la tabla 4.1. Un ejemplo de las mediciones mostradas en la línea de comandos se puede observar en la figura 4.2.

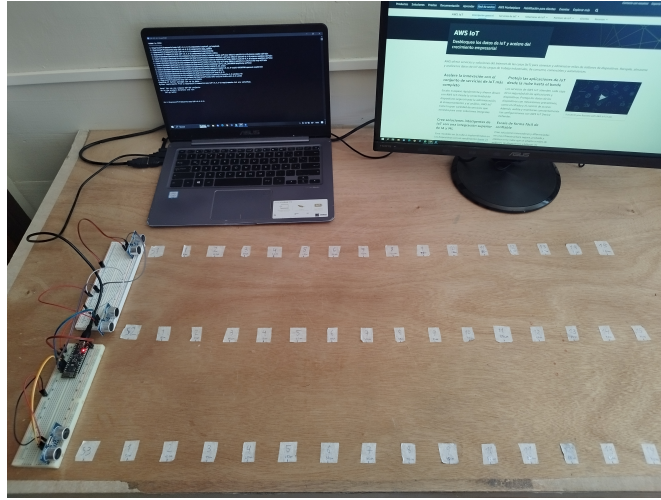


Figura 4.1: Entorno de pruebas para sensores y microcontrolador

Número Marca	Distancia real [cm]	Distancia medida sensor 1 [cm]	Estado notificado sensor 1	Distancia medida sensor 2 [cm]	Estado notificado sensor 2	Distancia medida sensor 3 [cm]	Estado notificado sensor 3
1	5	4	2	4	2	5	2
2	10	10	1	10	1	12	1
3	15	14	1	15	1	17	1
4	20	20	1	19	1	20	1
5	25	24	1	24	1	26	1
6	30	29	1	29	1	30	1
7	35	35	1	34	1	36	1
8	40	38	1	38	1	41	1
9	45	44	1	44	1	45	1
10	50	49	1	49	1	50	1
11	55	54	0	53	0	55	0
12	60	59	0	59	0	61	0
13	65	64	0	63	0	66	0
14	70	69	0	68	0	72	0
15	75	70	0	73	0	75	0

Tabla 4.1: Resultados de la prueba de medición de los sensores

Sean m los números de las marcas y s los números de los sensores, con $m = 1, 2, \dots, 20$ y $s = 1, 2, 3$, se tiene que el error porcentual del sensor s al medir la distancia del obstáculo en la marca m , denotado como E_{sm} esta dado por:

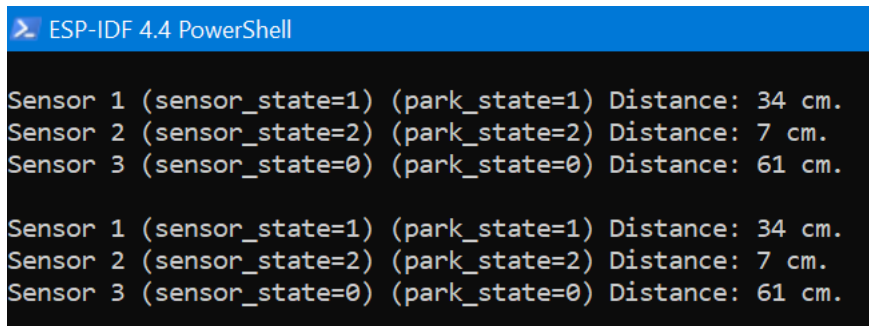
$$E_{sm} = \frac{|DM_{sm} - DR_m|}{|DR_m|} \cdot 100\%$$

donde DM_{sm} es la distancia medida por el sensor s al ubicar el obstáculo en la marca m y DR_m corresponde a la distancia real desde el sensor al obstáculo.

Una vez obtenidos todos los errores porcentuales E_{sm} , se calcula para cada sensor los errores porcentuales promedio para las mediciones de cada sensor (EP_s) obteniendo los siguientes resultados:

- $EP_1 = 3,76\%$
- $EP_2 = 3,89\%$
- $EP_3 = 3,25\%$

De estos resultados, se observa que todos los valores de error promedio no son mayores a 5% , por lo cual se cumple con lo especificado en el criterio de aceptación 1 de esta prueba.



```

ESP-IDF 4.4 PowerShell

Sensor 1 (sensor_state=1) (park_state=1) Distance: 34 cm.
Sensor 2 (sensor_state=2) (park_state=2) Distance: 7 cm.
Sensor 3 (sensor_state=0) (park_state=0) Distance: 61 cm.

Sensor 1 (sensor_state=1) (park_state=1) Distance: 34 cm.
Sensor 2 (sensor_state=2) (park_state=2) Distance: 7 cm.
Sensor 3 (sensor_state=0) (park_state=0) Distance: 61 cm.

```

Figura 4.2: Ejemplo de mediciones de los sensores mostrados en la línea de comandos

De la figura 4.2 se observa que las mediciones de los sensores y el estado de los estacionamientos aparece por pantalla en la terminal, por lo tanto esto cumple con el criterio de aceptación 2 definido anteriormente.

Con ambos criterios de aceptación cumplidos se puede afirmar que se cumplen los requisitos funcionales 1 y 2 del Subsistema de sensores y los requisitos 1 a 4 del Subsistema de Dispositivos de Borde.

4.2. Comunicación entre Microcontrolador AWS

Esta prueba es diseñada para comprobar si se cumplen el requisito funcional 5 del Subsistema de Dispositivos de Borde y el requisito funcional 1 del Subsistema de Nube de IoT. Estos están detallados en las secciones 2.3.2 y 2.3.3 respectivamente.

Para este fin se definen los siguientes criterios de aceptación

1. Si el cliente de prueba de AWS se suscribe al tópico “esp32/1”, recibe mensajes cuando alguno de los sensores actualiza su estado.
2. El contenido de los mensajes enviados a AWS es consistente con el estado de los sensores.

Haciendo uso del mismo entorno de pruebas de la sección 4.1 se realizan los siguientes pasos:

- Posicionar los sensores en el mismo entorno de pruebas utilizado en la sección 4.1 (observar figura 4.1).
- Iniciar sesión en consola de administración de AWS (por medio de su página web) e ingresar al servicio “AWS IoT Core”. En el menú principal de este buscar en la barra lateral izquierda el vínculo “Cliente de prueba de MQTT” y pulsar este.
- En la página que se despliega, en la sección “Suscribirse a un tema” rellenar el campo “Filtro de temas” con el valor “esp32/1”.
- Pulsar “Configuración adicional” lo cual desplegará más opciones, de estas cambiar la opción “Calidad del servicio” por “Calidad del servicio 1: el mensaje se entregará al menos una vez” y “Visualización de la carga de MQTT” por “Mostrar cargas como cadenas (más preciso)”.
- Una vez seleccionadas las opciones, pulsar el botón “Suscribir”. Esto permitirá que el cliente de prueba reciba los mensajes y despliegue en pantalla los mensajes enviados por el microcontrolador con el tópico “esp/32”.
- Ejecutar el código fuente del Subsistema de Dispositivos de Borde (sin modificaciones) en una línea de comandos, tal como se hace en la sección 4.1. Asegurar que no haya ningún obstáculo en ninguna de las etiquetas.
- Esperar a que el cliente de prueba reciba un mensaje que notifique que el estado de todos los sensores están como *Libre*, estos mensajes tienen como contenido

“puesto:1:0” , “puesto:2:0” y “puesto:3:0” para los puestos de estacionamiento 1, 2 y 3 respectivamente.

- Colocar un obstáculo en la etiqueta 1 del primer sensor (5) y observar si el cliente recibe un mensaje con el contenido “puesto:1:2” el cual confirma que el estado del sensor 1 ha cambiado a *Sucio*.
- Deslizar el obstáculo hasta la etiqueta 3 (15) y observar en el cliente el mensaje con el contenido “puesto:1:1” el cual confirma que el estado ha cambiado a *Ocupado*.
- Deslizar el obstáculo hasta la etiqueta 11 (55) y observar que el cliente es notificado del cambio al estado *Libre* con el contenido “puesto:1:0”.
- Repetir los pasos anteriores con el resto de los sensores.

En la figura 4.3 se observan los mensajes recibidos por el cliente de prueba de AWS:

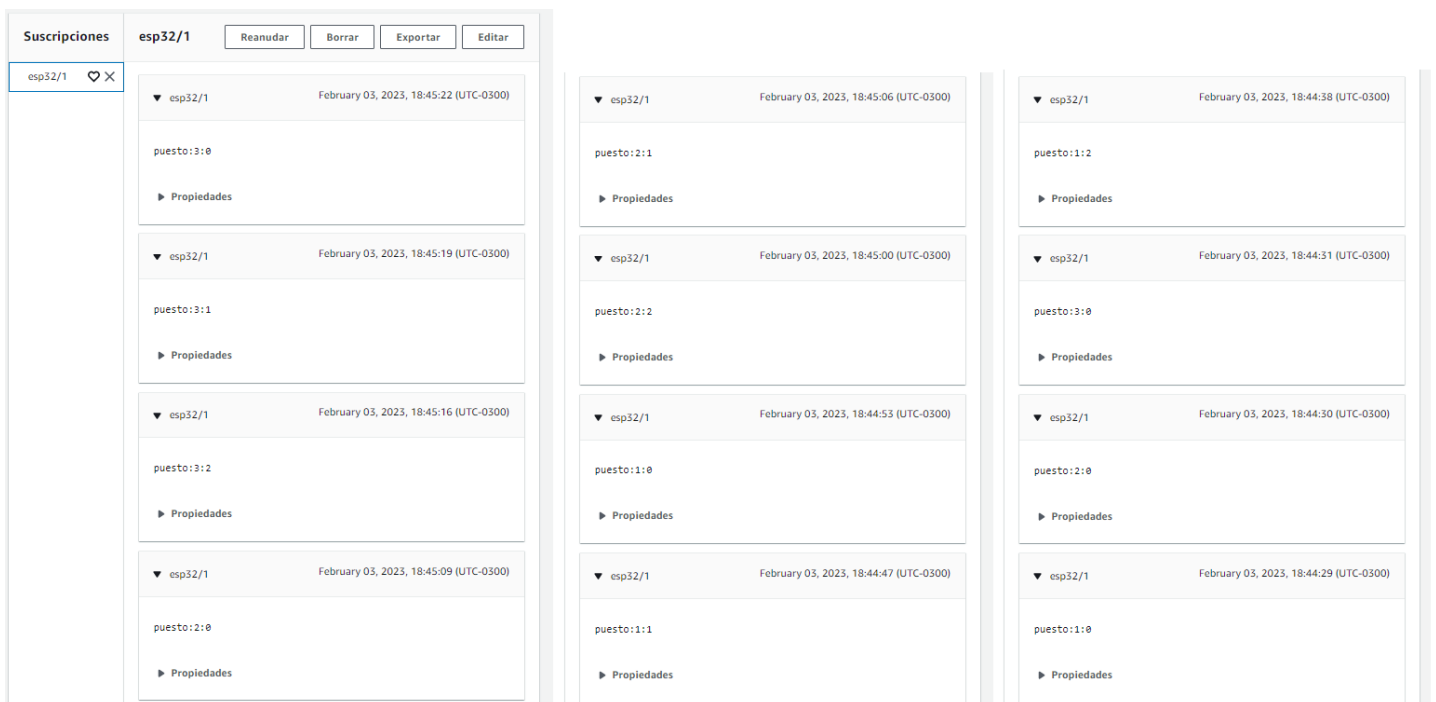


Figura 4.3: Mensajes del microcontrolador recibidos por el cliente de prueba de AWS

Como se observa, el cliente de prueba recibe todos los cambios de estado de los sensores de forma consistente, por lo tanto se cumplen los criterios de aceptación 1 y 2 de esta prueba.

Con los dos criterios de aceptación cumplidos se puede aseverar que los requisitos funcionales del Subsistema de Dispositivos de Borde y el requisito funcional 1 del Subsistema de Nube de IoT se cumplen.

4.3. Comunicación entre AWS y aplicación Android

La prueba diseñada a continuación, tiene como objetivo demostrar el correcto funcionamiento del requisito funcional 2 del Subsistema de Nube de IoT y los requisitos funcionales 1, 2 y 3 del subsistema de aplicación. Estos están especificados en las secciones 2.3.3 y 3.1.4 respectivamente.

Para esta prueba se definen los siguientes criterios de aceptación

1. Si el cliente de prueba de AWS publica mensajes con el tópico “esp32/1” con carga equivalente a la que envía el microcontrolador cuando ocurre un cambio de estado, la aplicación muestra el nuevo estado por pantalla.
2. La aplicación responde a los mensajes recibidos, independiente de la red en la que se encuentre conectado el dispositivo móvil.

Estos criterios se ponen a prueba realizando los siguientes pasos:

- Abrir aplicación de usuario, pulsar el botón “CONNECT”, una vez el texto junto a “Status:” aparezca como “Connected” pulsar el botón “SUBSCRIBE”. Con esto la aplicación se encuentra preparada para recibir mensajes de AWS.
- Abrir el cliente de prueba de MQTT de AWS (instrucciones en sección 4.2) y abrir la pestaña “Publicar en un tema” y rellenar el campo “Nombre del tema” con el valor “esp32/1”.
- En la opción “Calidad de servicio” dentro de la sección desplegable “Configuración adicional”, seleccionar “Calidad del servicio 1: el mensaje se entregará al menos una vez”.
- Para el campo “Carga del mensaje” escribir como contenido “puesto:1:2”, pulsar el botón “Publicar” y verificar en la aplicación que el estado del estacionamiento 1 cambie a *Sucio* (En interfaz de usuario aparece como *Verificar*).
- Repetir el paso anterior, pero utilizar como carga del mensaje el contenido “puesto:2:1”. Una vez publicado el mensaje verificar que el estado del estacionamiento 2 cambie a ocupado.
- Repetir el paso anterior, pero se utiliza como carga, el contenido “puesto:3:0”. Una vez enviado el mensaje verificar que el estado del estacionamiento 3 se mantiene como *Libre*.

- Si se repiten los pasos anteriores utilizando otra conexión a la red, por ejemplo desconectándose de la red Wifi local y utilizar la conexión por plan de datos, el comportamiento de la aplicación es el mismo.

En la figura 4.4 se observa el comportamiento de la aplicación ante los mensajes recibidos.

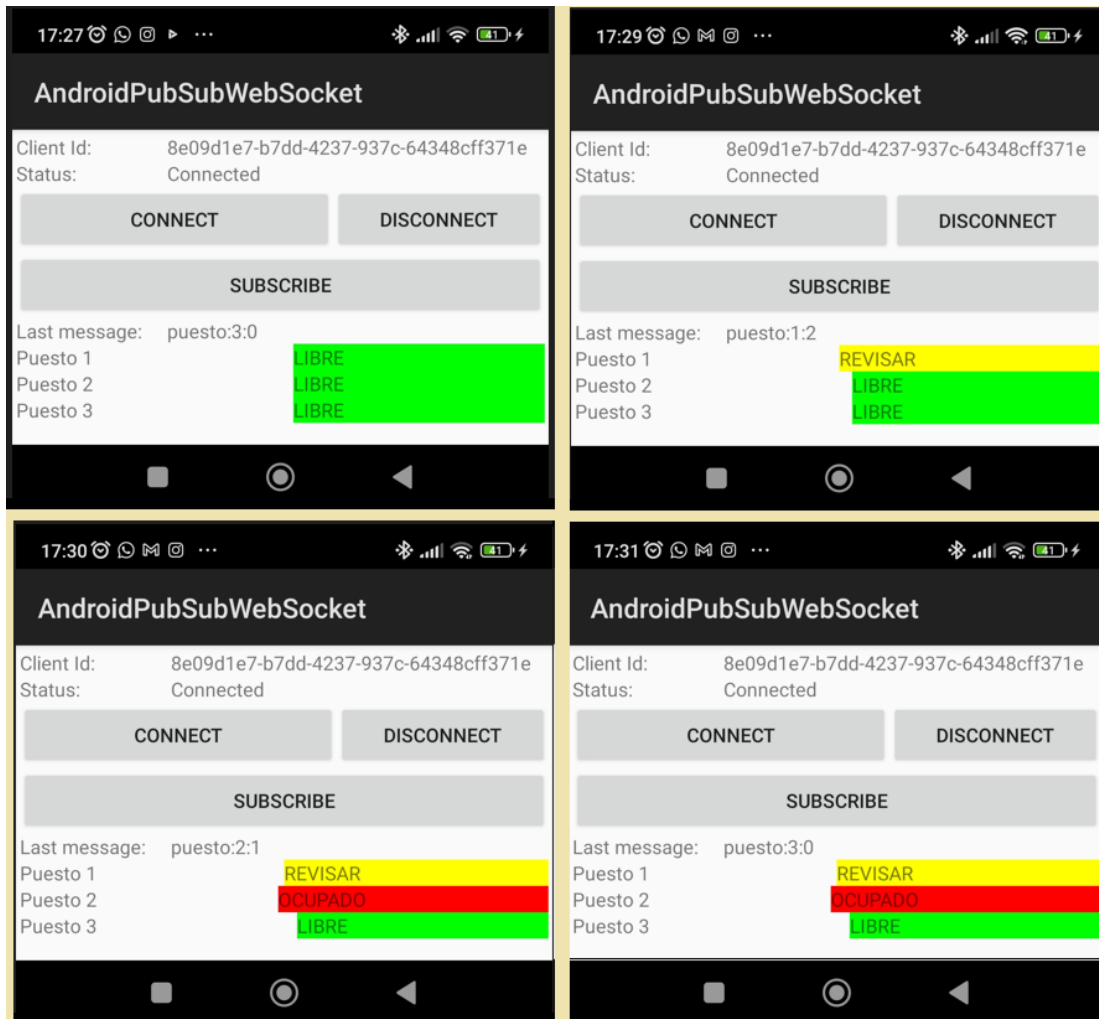


Figura 4.4: Comportamiento de la aplicación al recibir mensajes del cliente de prueba de AWS

Como se puede observar la aplicación responde a los mensajes enviados por el cliente de prueba de AWS por lo tanto se cumple el criterio de aceptación 1.

En la figura 4.5 se observa el comportamiento de la aplicación a aplicar las mismas pruebas, pero con el dispositivo móvil conectado a otra red. Al observar en la parte superior de la pantalla en la figura 4.4 el dispositivo móvil se encuentra conectado a

una red WiFi y en la figura 4.5 se encuentra conectado a la red móvil.

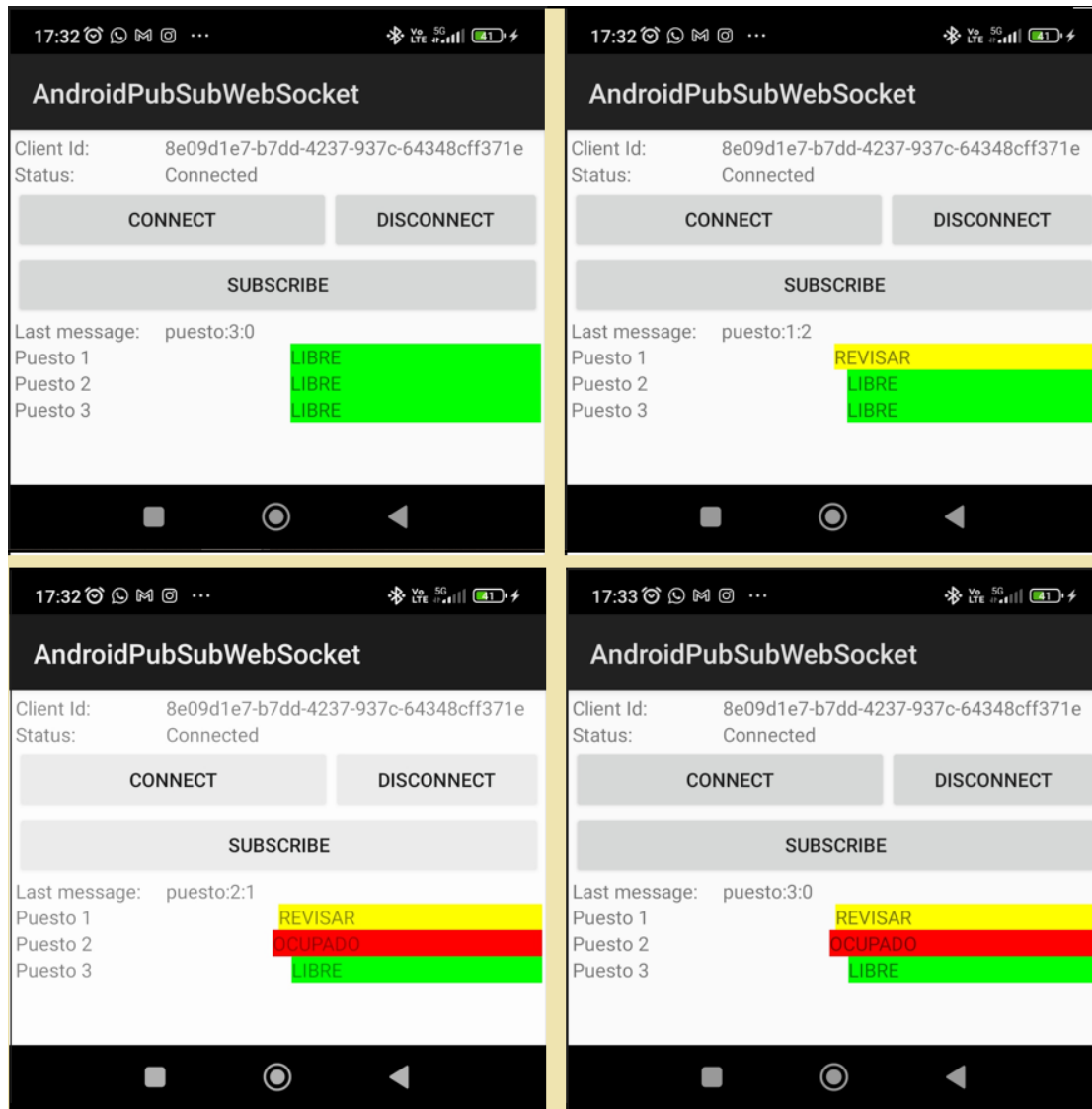


Figura 4.5: Comportamiento de la aplicación al recibir mensajes del cliente de prueba de AWS con el dispositivo móvil conectado a otra red.

Como se puede observar, el comportamiento de la aplicación se mantiene, a pesar de estar conectado a otra red de Internet, por lo tanto se cumple el criterio de aceptación 2.

Con todos los criterios de aceptación de esta prueba cumplidos, se comprueba que el requisito funcional 2 del Subsistema de Nube de IoT y los requisitos funcionales 1, 2 y 3 del Subsistema de Aplicación se cumplen.

4.4. Funcionamiento del sistema completo

Para comprobar que el sistema completo cumple con los requisitos funcionales definidos en la sección 2.2.1 se definen los siguientes criterios de aceptación:

1. Si se coloca un obstáculo entre las etiquetas 1 y 2, la aplicación notifica el estado del respectivo estacionamiento como “Sucio”.
2. Si se coloca un obstáculo entre las etiqueta 2 y 10, la aplicación notifica el estado del respectivo estacionamiento como “Ocupado”.
3. Si se coloca un obstáculo más alejado que la etiqueta 10 o no se coloca ningún obstáculo, la aplicación notifica el estado del respectivo estacionamiento como “Libre”.

Para la prueba a realizarse se utiliza el mismo entorno de pruebas que en la sección 4.1, a este se le añaden dos filas más de marcas ubicadas en paralelo a la ya existente, de tal manera de poder realizar pruebas con todos los sensores simultáneamente.

Una vez preparado el entorno de pruebas se realizan los siguientes pasos:

- Realizar las conexiones y ejecutar el código fuente, de la misma manera que se realiza en la sección 4.1. Asegurar que los sensores están en sus respectivas posiciones.
- Abrir la aplicación móvil y prepararla tal como se hace en la sección 4.3.
- Colocar un obstáculo en la etiqueta 1 (5 cm) del espacio del sensor 1. Verificar que la aplicación muestra por pantalla el estado del estacionamiento 1 como *Sucio* (aparece como *Verificar* en la interfaz gráfica).
- Colocar un obstáculo en las etiquetas 4 y 6 de los espacios de los sensores 2 y 3 respectivamente. Verificar que el estado de los estacionamientos 2 y 3 de la aplicación cambian a *Ocupado*.
- Retirar todos los obstáculos y verificar que todos los puestos de estacionamiento aparecen como *Libre*.
- Colocar un obstáculo en las etiquetas 8 de los espacios de todos los sensores. Verificar que todos los espacios de estacionamientos aparecen como *Ocupado*.

Para esta prueba se realiza un video demostrativo del funcionamiento del sistema el cual demuestra el cumplimiento de los criterios de aceptación 1 2 y 3, el cual puede ser observado en el enlace <https://youtu.be/0hD44QMT3AU>.

4.5. Medidas de desempeño

Para medir y/o estimar el desempeño del sistema se definen diferentes indicadores. A continuación se describe la nomenclatura de los indicadores y los procedimientos que se realizan para obtenerlos.

- Tiempo de latencia del sistema(t_l): Corresponde al tiempo transcurrido entre que un sensor ultrasonido comienza a realizar la medición que genera un cambio en un puesto de estacionamiento, hasta que la aplicación notifica al usuario del cambio. Debido a la complejidad para realizar esta medición, se separa este tiempo en tres subintervalos, denominados t_{me} , t_{rp} y t_{na} , tal que $t_l = t_{me} + t_{rp} + t_{na}$.
- Tiempo de medición y entrega (t_{me}): Corresponde al tiempo entre inicio de medición del sensor hasta entrega de medición a la cola. Este tiempo es medido con lógica en el código fuente.
- Tiempo de recibo y publicación(t_{rp}): Corresponde al tiempo entre recibo de medición de la cola y publicación de mensaje en AWS. Este tiempo es medido con lógica en el código fuente.
- Tiempo de nube a aplicación(t_{na}): Tiempo entre envío de mensaje desde AWS hasta cambio en aplicación de usuario. Para medir este tiempo se graba un vídeo (disponible en el enlace <https://youtu.be/iOvTa05u9nA>) el cual muestra simultáneamente el cliente de prueba de AWS y la aplicación de usuario, donde el cliente está siendo utilizado para enviar mensajes a la aplicación. Luego observando el video cuadro por cuadro, se determina el tiempo transcurridos entre el envío del mensaje y el cambio en la aplicación.

Cada uno de los procedimientos para estimar los tiempos definidos anteriormente se realizan un total de 20 mediciones y se obtiene el promedio de los valores obtenidos y los valores máximos y mínimos obtenidos. En la tabla 4.2 se observan los resultados obtenidos de estas mediciones y los valores promedio, mínimos y máximos.

Medición	t_me (s)	t_rp (s)	t_ma (s)
1	0,02	0,3	0,387
2	0	0,41	0,161
3	0,02	0,33	0,129
4	0	0,25	0,161
5	0,01	0,28	0,258
6	0	0,29	0,194
7	0,02	0,22	0,161
8	0	0,35	0,194
9	0,01	0,27	0,161
10	0	0,4	0,226
11	0,01	0,32	0,161
12	0	0,23	0,161
13	0,02	0,36	0,29
14	0	0,28	0,194
15	0,01	0,4	0,129
16	0	0,32	0,194
17	0,01	0,24	0,29
18	0,02	0,35	0,129
19	0	0,27	0,29
20	0	0,19	0,194

	t_me (s)	t_rp (s)	t_ma (s)
Promedio	0,0075	0,303	0,2032
Mínimo	0	0,19	0,129
Máximo	0,02	0,41	0,387

Tabla 4.2: Resultados de mediciones de desempeño

En base a los resultados de la tabla, al sumar los valores promedio de t_{me} , t_{rp} y t_{na} , se obtiene que el tiempo promedio de latencia es de 0,5137 segundos. Observando los tiempos mínimos y máximos al añadirlos se obtiene que en el caso más favorable el tiempo de latencia del sistema es de 0,148 segundos y en el caso menos favorable de 0,817 segundos.

Con estos valores de desempeño temporal, se puede afirmar que el sistema actúa en tiempo real, ya que a percepción humana el cambio en la aplicación de usuario ocurre de manera prácticamente instantánea. Además esto cumple con el requisito no funcional 2 del sistema completo.

Capítulo 5

Conclusiones y trabajo futuro

En base a lo mencionado en las secciones anteriores, y con los requisitos funcionales y no-funcionales del sistema satisfechos, se puede afirmar que este cumple con los objetivos planteados en la sección 1.3. Por lo tanto, se demuestra que con ayuda de los servicios de AWS en la nube es factible crear un proyecto de IoT con dispositivos y sensores de bajo costo y recursos limitados.

A pesar de los logros alcanzados, aún es necesario realizar mejoras al sistemas si se desea hacer uso de este en un entorno real. Para mejorar la robustez del sistema, destaca la necesidad de estudiar los códigos de error de las librerías utilizadas para el desarrollo del proyecto para lograr que el sistema posea un mejor manejo ante situaciones de error. Crear archivos de configuración para dar valores a las diferentes constantes del código fuente sería una mejora de gran ayuda para facilitar la adaptabilidad del sistema a diferentes tales como la naturaleza de los estacionamientos, cantidad de dispositivos de borde necesarios para abarcar los estacionamientos de un lugar o recinto, entre otros. Los archivos de configuración son un factor que facilita también la escalabilidad del sistema. Otro factor importante es la necesidad de proteger la integridad de los sensores para evitar que sean aplastados y destruidos por los neumáticos de los vehículos. Una posible solución a este problema es la creación de una carcasa protectora que sea capaz de resistir el peso del vehículo y, al mismo tiempo, no interfiera con las mediciones del sensor. Una propuesta de protección para los sensores se observa en el anexo D.

Por otro lado, cualquier instalación de sensores en terreno público implica negociaciones y autorizaciones de las autoridades locales. Si esto no se realiza exitosamente, supone un impedimento para un proyecto de instalación del SMPE en la vía pública.

Es importante considerar este factor y realizar las gestiones necesarias para obtener los permisos correspondientes.

Uno de los mayores desafíos presentes en la implementación del sistema SMPE es la configuración del framework esp-idf y lograr la correcta comunicación con los servicios de AWS en la nube. Para esto se dedican extensas jornadas de resolución de problemas por errores de diferentes índoles, entre ellos errores de configuración, permisos, certificados, versiones, entre otros.

Un sistema de monitoreo de puestos de estacionamiento utilizando servicios de IoT de AWS es solamente un ejemplo de los muchos proyectos que se pueden implementar en diversas áreas haciendo uso de estos servicios. AWS ofrece una amplia gama de opciones y herramientas para la implementación de soluciones IoT, desde la creación de flujos de datos hasta la integración con otros servicios en la nube. La variedad de dispositivos y sensores que se pueden utilizar es muy amplia, y se pueden conectar a la nube de AWS para gestionar y analizar los datos que producen. Por lo tanto, las posibilidades son ilimitadas y todo depende de la creatividad y necesidades específicas de cada proyecto.

Bibliografía

- [1] Hays. Una mirada a la Transformación Digital en la industria [en línea]<<https://www.hays.cl/contenido/blog-consejos-de-carrera/insights/la-transformacion-digital-en-la-industria>>[consulta: 19 febrero 2023]
- [2] Oracle. ¿Qué es el IoT? [en línea] <<https://www.oracle.com/cl/internet-of-things/what-is-iot/>>[consulta: 19 febrero 2023]
- [3] Iberdrola. ¿Qué es el IIoT? Descubre el Internet Industrial de las Cosas [en línea] <<https://www.iberdrola.com/innovacion/que-es-iiot> >[consulta: 19 febrero 2023]
- [4] Amazon. Sitio oficial de Amazon Web Services [en línea] <<https://aws.amazon.com/es/>>[consulta: 24 febrero 2023]
- [5] Amazon Web Services. Sitio informativo del servicio AWS IoT Core [en línea] <<https://aws.amazon.com/es/iot-core/>>[consulta: 19 febrero 2023]
- [6] Microsoft Azure. Sitio informativo de Microsoft Azure [en línea] <<https://azure.microsoft.com/es-es/> [consulta: 24 febrero 2023]
- [7] Microsoft Azure. Sitio informativo del servicio Azure IoT Hub [en línea] <<https://azure.microsoft.com/en-us/products/iot-hub/>>[consulta: 19 febrero 2023]
- [8] Google Cloud. Sitio informativo del servicio IoT Core de Google Cloud [en línea] <<https://cloud.google.com/iot-core?hl=es-419>>[consulta: 19 febrero 2023]
- [9] Cadem. Más del 70% de los hogares en Chile tiene automóvil y 30% de las personas prevé comprar uno este año [en línea] <<https://cadem.cl/mas-del-70-de-los-hogares-en-chile-tiene-automovil-y-30-de-las-personas-preve-comprar-uno-este-ano/> >[consulta: 19 febrero 2023]

- [10] Mobidev. IoT Based Smart Parking System Development [en línea] <<https://mobidev.biz/blog/iot-based-smart-parking-system> >[consulta: 20 febrero 2023]
- [11] Sparkfun. Datasheet de sensor ultrasonido HC-SR04 [en línea] <<https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf> >[consulta: 24 febrero 2023]
- [12] Espressif. Sitio descriptivo del microcontrolador ESP8266 [en línea] <<https://www.espressif.com/en/products/socs/esp8266> >[consulta: 27 febrero 2023]
- [13] Amazon Web Services. Sitio informativo del servicio AWS Lambda [en línea] <<https://aws.amazon.com/es/lambda/> >[consulta: 20 febrero 2023]
- [14] Amazon Web Services. Sitio informativo del servicio AWS S3 [en línea] <<https://aws.amazon.com/es/s3/> >[consulta: 24 febrero 2023]
- <https://mqtt.org/>
- [15] AUTOR. Sitio oficial del protocolo MQTT [en línea] <<https://mqtt.org/> >[consulta: 20 febrero 2023]
- [16] Shoban Kumar. Repositorio de proyecto de Smart Parking de Shoban Kumar [en línea] <<https://github.com/shobankr/IoT-Suite-Smart-parking> >[consulta: 24 febrero 2023]
- [17] Empresa Raspberry Pi. Página oficial de Raspberry Pi 2 Model B [en línea] <<https://www.raspberrypi.com/products/raspberry-pi-2-model-b/> >[consulta: 24 febrero 2023]
- [18] Documentación de Smooch API [en línea] <<https://docs.smooch.io/rest/#section/Introduction> >[consulta: 24 febrero 2023]
- [19] Slack. Sitio oficial de Slack [en línea] <<https://slack.com/intl/es-cl/> >[consulta: 24 febrero 2023]
- [20] Microsoft. Documentación oficial de Microsoft Windows 10 IoT Core [en línea] <<https://learn.microsoft.com/es-es/windows/iot-core/windows-iot-core> >[consulta: 24 febrero 2023]

- [21] Datasheet de sensor ultrasonido HY-SRF05 [en línea] <<https://datasheetspdf.com/pdf-file/813041/ETC/HY-SRF05/1>>[consulta: 24 febrero 2023]
- [22] Espressif. Documentación oficial de las placas ESP32-PICO-KIT versiones 4 y 4.1 [en línea] <<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-pico-kit.html#get-started-pico-kit-v4-board-front>>[consulta: 24 febrero 2023]
- [23] Amazon Web Services. Catálogo de dispositivos compatibles con AWS IoT Core [en línea] <<https://devices.amazonaws.com/search?page=1sv=iot>>[consulta: 24 febrero 2023]
- [24] Altronics. Página de venta de sensor HY-SRF05 [en línea] <<https://altronics.cl/sensor-ultrasonico-hc-sr05>>[consulta: 25 febrero 2023]
- [25] ST. Página de venta del dispositivo STM32L4 Discovery Kit IoT Node [en línea] <<https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html>>[consulta: 25 febrero 2023]
- [26] Microchip. Página de venta del dispositivo PIC-IoT Development Board [en línea] <<https://www.microchipdirect.com/dev-tools/EV54Y39A?allDevTools=true>>[consulta: 25 febrero 2023]
- [27] Microchip. Página de venta del dispositivo SAM E54 Xplained Pro [en línea] <<https://www.microchip.com/en-us/development-tool/atsame54-xpro#>>[consulta: 25 febrero 2023]
- [28] Empresa Raspberry Pi. Página oficial de Raspberry Pi 3 Model B [en línea] <<https://www.raspberrypi.com/products/raspberry-pi-3-model-b/>>[consulta: 25 febrero 2023]
- [29] Mouser. Sitio de compra del dispositivo ESP32-PICO-KIT [en línea] <<https://www.mouser.cl/ProductDetail/Espressif-Systems/ESP32-PICO-KIT?qs=MLItCLRbWsyoLrlknFRqcQ%3D%3D>>[consulta: 25 febrero 2023]
- [30] Espressif. Sitio informativo del framework esp-idf [en línea] <<https://www.espressif.com/en/products/sdks/esp-idf>>[consulta: 25 febrero 2023]

- [31] Amazon Web Services. Sitio informativo de Amazon FreeRTOS [en línea] <<https://aws.amazon.com/es/freertos/> >[consulta: 25 febrero 2023]
- [32] Amazon Web Services. Sitio informativo del servicio AWS IoT Device Management [en línea] <<https://aws.amazon.com/es/iot-device-management/> >[consulta: 25 febrero 2023]
- [33] Amazon Web Services. Sitio informativo del servicio Amazon Cognito [en línea] <<https://aws.amazon.com/es/cognito/> >[consulta: 25 febrero 2023]
- [34] Android. Sitio oficial de Android Studio [en línea] <<https://developer.android.com/studio> >[consulta: 25 febrero 2023]
- [35] Github. Repositorio “Android PubSub with WebSockets Sample” de Felipe Meriga [en línea] <<https://github.com/felipemeriga/aws-sdk-android-samples/tree/master/AndroidPubSubWebSocket> >[consulta: 27 febrero 2023]
- [36] Amazon Web Services. Instrucciones de instalación y configuración de esp-idf [en línea] <https://docs.aws.amazon.com/freertos/latest/userguide/getting_started_espessif.html>[consulta: 27 febrero 2023]
- [37] Github. Repositorio de Amazon FreeRTOS [en línea] <<https://github.com/aws/amazon-freertos> >[consulta: 25 febrero 2023]
- [38] Espressif. Sitio de descarga del instalador del framework ESP-IDF [en línea] <<https://dl.espressif.com/dl/esp-idf/?idf=4.4> >[consulta: 25 febrero 2023]
- [39] Artículo descriptivo del uso de archivos CMakeLists [en línea] <<https://programmerclick.com/article/6282707862/> >[consulta: 25 febrero 2023]
- [40] Github. Código fuente de proyecto de ejemplo subscribe_publish [en línea] <https://github.com/espressif/esp-aws-iot/tree/release/v3.1.x/examples/subscribe_publish >[consulta: 25 febrero 2023]
- [41] FreeRTOS. Documentación de la función xQueueCreate de FreeRTOS [en línea] <<https://www.freertos.org/a00116.html> >[consulta: 25 febrero 2023]

- [42] FreeRTOS. Documentación de la función `xTaskCreate` de FreeRTOS [en línea] <<https://www.freertos.org/a00125.html> >[consulta: 25 febrero 2023]
- [43] Espressif. Documentación del driver “ultrasonic” de esp-idf [en línea] <<https://esp-idf-lib.readthedocs.io/en/latest/groups/ultrasonic.html> >[consulta: 25 febrero 2023]
- [44] FreeRTOS. Documentación de la función `xQueueSend` de FreeRTOS [en línea] <<https://www.freertos.org/a00117.html> >[consulta: 25 febrero 2023]
- [45] Amazon Web Services. Documentación de la estructura “`AWS_IoT_Client`” [en línea] <http://aws-iot-device-sdk-embedded-c-docs.s3-website-us-east-1.amazonaws.com/struct__Client.html >[consulta: 25 febrero 2023]
- [46] Amazon Web Services. Documentación de la estructura “`IoT_Client_Init_Params`” [en línea] <http://aws-iot-device-sdk-embedded-c-docs.s3-website-us-east-1.amazonaws.com/structIoT__Client__Init__Params.html >[consulta: 25 febrero 2023]
- [47] FreeRTOS. Documentación de la función `xEventGroupWaitBits` de FreeRTOS [en línea] <<https://www.freertos.org/xEventGroupWaitBits.html> >[consulta: 25 febrero 2023]
- [48] FreeRTOS. Documentación de la función `xEventGroupSetBits` de FreeRTOS [en línea] <<https://www.freertos.org/xEventGroupSetBits.html> >[consulta: 25 febrero 2023]
- [49] Amazon Web Services. Documentación de la estructura “`IoT_Client_Connect_Params`” [en línea] <http://aws-iot-device-sdk-embedded-c-docs.s3-website-us-east-1.amazonaws.com/structIoT__Client__Connect__Params.html >[consulta: 25 febrero 2023]
- [50] Amazon Web Services. Documentación de la función `aws_iot_mqtt_connect` [en línea] <http://aws-iot-device-sdk-embedded-c-docs.s3-website-us-east-1.amazonaws.com/aws__iot__mqtt__client__connect_8c.html#aa379a6469c9a6aefc52d4c00a2b37567 >[consulta: 25 febrero 2023]

- [51] Amazon Web Services. Documentación de la función `aws_iot_mqtt_autoreconnect_set_status` [en línea] <IoT_Publish_Message_Param >[consulta: 25 febrero 2023]
- [52] Amazon Web Services. Documentación de la estructura `IoT_Publish_Message_Params` [en línea] <http://aws-iot-device-sdk-embedded-c-docs.s3-website-us-east-1.amazonaws.com/structIoT_Publish_Message_Params.html >[consulta: 25 febrero 2023]
- [53] Amazon Web Services. Lista de códigos de respuesta de la enumeración `IoT_Error_t` [en línea] <http://aws-iot-device-sdk-embedded-c-docs.s3-website-us-east-1.amazonaws.com/aws_iot_error_8h.html#a329da5c3a42979b72561e28e749f6921 >[consulta: 25 febrero 2023]
- [54] Amazon Web Services. Documentación de la función `aws_iot_mqtt_yield` [en línea] <http://aws-iot-device-sdk-embedded-c-docs.s3-website-us-east-1.amazonaws.com/aws_iot_mqtt_client_yield_8c.html#a0b638ebb7859f459be2de2fbb7419aa0 >[consulta: 25 febrero 2023]
- [55] Amazon Web Services. ¿Cómo creo y activo una nueva cuenta de AWS? [en línea] <<https://aws.amazon.com/es/premiumsupport/knowledge-center/create-and-activate-aws-account/> >[consulta: 25 febrero 2023]
- [56] Computer Hoy. Qué son los UUID y por qué son útiles [en línea] <<https://computerhoy.com/reportajes/tecnologia/son-uuid-son-utiles-1026215> >[consulta: 25 febrero 2023]
- [57] AUTOR. NOMBRE ARTICULO [en línea] <LINK >[consulta: 25 febrero 2023]

Anexo A

Código fuente de smart_parking.c

```
/*
 * smart_parking.c
 *
 * Created on: 02-12-2022
 * Author: Esteban
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <unistd.h>
#include <limits.h>
#include <string.h>
#include <time.h>

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/event_groups.h"
#include "esp_system.h"
#include "esp_wifi.h"
#include "esp_event.h"
#include "esp_log.h"
#include "esp_vfs_fat.h"
#include "driver/sdmmc_host.h"

#include "nvs.h"
#include "nvs_flash.h"

#include "aws_iot_config.h"
#include "aws_iot_log.h"
#include "aws_iot_version.h"
#include "aws_iot_mqtt_client_interface.h"

#include <esp_idf_version.h>
#include "driver/gpio.h"
#include "ultrasonic.h"

#if ESP_IDF_VERSION >= ESP_IDF_VERSION_VAL(4, 1, 0)
// Features supported in 4.1+
#define ESP_NETIF_SUPPORTED
#endif

static SemaphoreHandle_t bin_sem;
static const char *TAG = "subpub";
// static int send_message = false;
// static int gpio_num = 0;
```

```

// Definiciones para GPIO
#define GPIO_INPUT_IO_0 21
#define GPIO_INPUT_IO_1 22
#define GPIO_INPUT_IO_2 19
#define GPIO_INPUT_IO_3 23
#define GPIO_INPUT_IO_4 18
#define GPIO_INPUT_IO_5 5
#define GPIO_INPUT_IO_6 10
#define GPIO_INPUT_IO_7 9
#define GPIO_INPUT_PIN_SEL ((1ULL<<GPIO_INPUT_IO_0) | (1ULL<<GPIO_INPUT_IO_1) | (1ULL<<GPIO_INPUT_IO_2) | \
(1ULL<<GPIO_INPUT_IO_3) | (1ULL<<GPIO_INPUT_IO_4) | (1ULL<<GPIO_INPUT_IO_5) | (1ULL<<GPIO_INPUT_IO_6) | \
(1ULL<<GPIO_INPUT_IO_7))
#define ESP_INTR_FLAG_DEFAULT 0

// Definiciones para sensores ultrasonido
#define NUMBER_OF_SENSORS 3
#define GPIO_TRIGGER_0 21
#define GPIO_ECHO_0 22
#define GPIO_TRIGGER_1 19
#define GPIO_ECHO_1 23
#define GPIO_TRIGGER_2 18
#define GPIO_ECHO_2 5
#define GPIO_TRIGGER_3 10
#define GPIO_ECHO_3 9
#define MAX_DISTANCE_CM 300 // 2m max

// Definición estados estacionamientos
#define FREE 0
#define OCCUPIED 1
#define DIRTY 2
#define INIT 3

// Definiciones varias
#define TRUE 1
#define FALSE 0
#define UPDATE_DATA 200

typedef struct Sensors_info{
    char state[NUMBER_OF_SENSORS];
    char updated[NUMBER_OF_SENSORS];
    char to_change;
} Sensors_info;

// QueueHandle_t gpio_evt_queue = NULL;
QueueHandle_t aws_iot_evt_queue = NULL;

// Definiciones para conexión a Wifi
// #define WIFI_SSID "riparia-2g"
// #define WIFI_PASS "riparia49"
#define WIFI_SSID "Redmi_Note_11_Pro_5G"
#define WIFI_PASS "holahola"

/* FreeRTOS event group to signal when we are connected & ready to make a request */
static EventGroupHandle_t wifi_event_group;

/* The event group allows multiple bits for each event,
but we only care about one event - are we connected
to the AP with an IP? */
const int CONNECTED_BIT = BIT0;

// Cargando certificados
extern const uint8_t aws_root_ca_pem_start[] asm("_binary_aws_root_ca_pem_start");
extern const uint8_t aws_root_ca_pem_end[] asm("_binary_aws_root_ca_pem_end");
extern const uint8_t certificate_pem crt_start[] asm("_binary_certificate_pem crt_start");
extern const uint8_t certificate_pem crt_end[] asm("_binary_certificate_pem crt_end");
extern const uint8_t private_pem_key_start[] asm("_binary_private_pem_key_start");
extern const uint8_t private_pem_key_end[] asm("_binary_private_pem_key_end");

// Punto de enlace a cuenta de AWS

```

```

char HostAddress[255] = "at0bms8uo1fts-ats.iot.us-east-1.amazonaws.com";

/**
 * @brief Default MQTT port is pulled from the aws-iot-config.h
 */
uint32_t port = AWS_IOT_MQTT_PORT;

static void wifi_event_handler(void* arg, esp_event_base_t event_base,
                               int32_t event_id, void* event_data)
{
    if (event_base == WIFILEVENT && event_id == WIFILEVENT_STA_START) {
        esp_wifi_connect();
    } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t* event = (ip_event_got_ip_t*) event_data;
        ESP_LOGI(TAG, "Connected with IP Address:" IPSTR, IP2STR(&event->ip_info.ip));
        /* Signal main application to continue execution */

        xEventGroupSetBits(wifi_event_group, CONNECTED_BIT);
    } else if (event_base == WIFILEVENT && event_id == WIFILEVENT_STA_DISCONNECTED) {
        ESP_LOGI(TAG, "Disconnected. Connecting to the AP again...");
        esp_wifi_connect();
        xEventGroupClearBits(wifi_event_group, CONNECTED_BIT);
    }
}

void disconnectCallbackHandler(AWS_IoT_Client *pClient, void *data) {
    ESP_LOGW(TAG, "MQTT Disconnect");
    IoT_Error_t rc = FAILURE;

    if(NULL == pClient) {
        return;
    }

    if(aws_iot_is_autoreconnect_enabled(pClient)) {
        ESP_LOGI(TAG, "Auto_Reconnect_is_enabled, Reconnecting attempt will start now");
    } else {
        ESP_LOGW(TAG, "Auto_Reconnect_not_enabled, Starting manual reconnect...");
        rc = aws_iot_mqtt_attempt_reconnect(pClient);
        if(NETWORK_RECONNECTED == rc) {
            ESP_LOGW(TAG, "Manual_Reconnect_Successful");
        } else {
            ESP_LOGW(TAG, "Manual_Reconnect_Failed--%d", rc);
        }
    }
}

short* map_sensor(short sensor_num){
    short* return_array = malloc(2*sizeof(short));
    switch (sensor_num)
    {
        case 0: return_array[0] = GPIO_TRIGGER_0; break;
        case 1: return_array[0] = GPIO_TRIGGER_1; break;
        case 2: return_array[0] = GPIO_TRIGGER_2; break;
        case 3: return_array[0] = GPIO_TRIGGER_3; break;
        default:
            ESP_LOGE(TAG, "ERROR: _MAPEANDO_PIN_NO_USADO");
            abort();
            break;
    }

    switch (sensor_num)
    {
        case 0: return_array[1] = GPIO_ECHO_0; break;
        case 1: return_array[1] = GPIO_ECHO_1; break;
        case 2: return_array[1] = GPIO_ECHO_2; break;
        case 3: return_array[1] = GPIO_ECHO_3; break;
        default:
            ESP_LOGE(TAG, "ERROR: _MAPEANDO_PIN_NO_USADO");
            abort();
    }
}

```

```

        break;
    }

    return return_array;
}

char get_park_state(float distance_cm) {
    if(distance_cm > 50){
        return FREE;
    }
    else if(distance_cm >= 10){
        return OCCUPIED;
    }
    else{
        return DIRTY;
    }
}

void aws_iot_task(void *param) {

    //Iniciando variables
    char cPayload[25];
    clock_t start, stop;
    IoT_Error_t rc = NETWORK_RECONNECTED;

    //Creación de cliente de AWS IoT
    AWS_IoT_Client client;

    //Parametros de Mqtt
    IoT_Client_Init_Params mqttInitParams = iotClientInitParamsDefault;
    mqttInitParams.enableAutoReconnect = false; // We enable this later below
    mqttInitParams.pHostURL = HostAddress;
    mqttInitParams.port = port;
    mqttInitParams.pRootCALocation = (const char *)aws_root_ca_pem_start;
    mqttInitParams.pDeviceCertLocation = (const char *)certificate_pem_cert_start;
    mqttInitParams.pDevicePrivateKeyLocation = (const char *)private_pem_key_start;
    mqttInitParams.mqttCommandTimeout_ms = 20000;
    mqttInitParams.tlsHandshakeTimeout_ms = 5000;
    mqttInitParams.isSSLHostnameVerify = true;
    mqttInitParams.disconnectHandler = disconnectCallbackHandler;
    mqttInitParams.disconnectHandlerData = NULL;

    rc = aws_iot_mqtt_init(&client, &mqttInitParams);
    if(SUCCESS != rc) {
        ESP_LOGE(TAG, "aws_iot_mqtt_init returned error: %d", rc);
        abort();
    }

    /* Esperar a que el WiFi aparezca como conectado */
    xEventGroupWaitBits(wifi_event_group, CONNECTED_BIT,
        false, true, portMAX_DELAY);

    //Parametros de conexión
    IoT_Client_Connect_Params connectParams = iotClientConnectParamsDefault;
    connectParams.keepAliveIntervalInSec = 60;
    connectParams.isCleanSession = true;
    connectParams.MQTTVersion = MQTT_3_1_1;
    connectParams.pClientID = "ESP32_for_SmartParking_2";
    connectParams.clientIDLen = (uint16_t) strlen("ESP32_for_SmartParking_2");
    connectParams.isWillMsgPresent = false;

    //Conectandose a AWS
    ESP_LOGI(TAG, "Connecting to AWS.....");
    do {
        rc = aws_iot_mqtt_connect(&client, &connectParams);
        if(SUCCESS != rc) {
            ESP_LOGE(TAG, "Error(%d) connecting to %s:%d", rc, mqttInitParams.pHostURL, mqttInitParams.port);
            vTaskDelay(1000 / portTICK_PERIOD_MS);
        }
    } while (SUCCESS != rc);
}

```

```

//Habilitar la autoreconexi n del cliente a AWS
rc = aws_iot.mqtt.autoreconnect_set_status(&client , true);
if (SUCCESS != rc) {
    ESP_LOGE(TAG, "Unable to set Auto-Reconnect to true - %d", rc);
    abort();
}

//T pico de mensajes a ser enviados
const char *TOPIC = "esp32/1";
const int TOPIC_LEN = strlen(TOPIC);

//Ajustes de mensajes
IoT_Publish_Message_Params paramsQOS1;
paramsQOS1.qos = QOS1;
paramsQOS1.payload = (void *) cPayload;
paramsQOS1.isRetained = 0;

Sensors.info* sensors_info;

ESP_LOGI(TAG,"Conexion exitosa\n");

//rc = UPDATE_DATA;

//Este loop se mantiene iterando siempre que ningun mensaje falle
while((NETWORK_ATTEMPTING_RECONNECT == rc || NETWORK_RECONNECTED == rc || SUCCESS == rc /*|| UPDATE_DATA == rc*/) ) {

    //Verificar el estado de conexi n con AWS, esperar por un m ximo de 100 ms
    rc = aws_iot.mqtt.yield(&client , 100);
    if (NETWORK_ATTEMPTING_RECONNECT == rc) {
        // Si el cliente est intentando reconectarse, continuar en la siguiente iteraci n
        continue;
    }

    //xSemaphoreTake(bin_sem , portMAX_DELAY);
    //ESP_LOGI(TAG,"Empezando iteracion: %d",rc);

    if (SUCCESS == rc){

        ESP_LOGI(TAG,"Accediendo a la cola aws-iot");

        //La cola recibe un
        if (xQueueReceive(aws_iot_evt_queue , &sensors_info , portMAX_DELAY)) {

            start = clock();
            //Preparando carga del mensaje
            short to_change_index = sensors_info->to_change;
            sprintf(cPayload, "%s:%d:%d", "puesto", sensors_info->to_change+1, sensors_info->state[to_change_index]);
            ESP_LOGI(TAG,"payload: %s",cPayload);
            paramsQOS1.payloadLen = strlen(cPayload);

            //Publicando mensaje
            //xSemaphoreTake(bin_sem , portTICK_PERIOD_MS*100);

            rc = aws_iot.mqtt.publish(&client , TOPIC, TOPIC_LEN, &paramsQOS1);
            stop = clock();
            //xSemaphoreGive(bin_sem);

            //En caso de xito se marca sensor como actualizado y se continua iterando
            if (rc == SUCCESS){
                sensors_info->updated[to_change_index] = TRUE;
                ESP_LOGI(TAG,"Mensaje publicado a aws exitosamente: %3f s",((double)(stop - start))/CLOCKS_PER_SEC);
                //rc = SUCCESS;
            }

            //Error: No se recibe el mensaje
            else if (rc == MQTT_REQUEST_TIMEOUT_ERROR) {
                ESP_LOGW(TAG, "QOS1-publish_ack_not_received.");
                short num_of_timeout_errors = 0;
                while(num_of_timeout_errors != 3){
                    rc = aws_iot.mqtt.publish(&client , TOPIC, TOPIC_LEN, &paramsQOS1);
                    if (rc == SUCCESS) {

```

```

        sensors_info->updated[to_change_index] = TRUE;
        ESP_LOGI(TAG, "Mensaje_enviado_a_aws_exitosamente: %.3f_s", ((double)(stop - start))/CLOCKS_PER_SEC);
        break;
    }
    else if (rc == MQTT_REQUEST_TIMEOUT_ERROR){
        num_of_timeout_errors += 1;
    }
    else{
        abort();
    }
}
abort();
}

else if (rc == NETWORK_SSL_READ_ERROR){
    ESP_LOGW(TAG, "ADVERTENCIA, _RETURN_CODE_-12");
    rc = SUCCESS;
}

//Cualquier otro error se reinicia microcontrolador
else {
    ESP_LOGE(TAG, "Problema_al_enviar_el_mensaje: %d", rc);
    abort();
}
}
else{
    ESP_LOGW(TAG, "aws_iot_no_recibio_ningun_mensaje_en_laCola\n");
}
//ESP_LOGI(TAG, "Pasando por aqui");
}

//Cuando red vuelve a conectarse actualizar los datos
else if (NETWORK_RECONNECTED == rc){
    if (sensors_info != NULL){
        ESP_LOGI(TAG, "Conexion_recuperada, _actualizando_datos");
        for (short i=0; i<NUMBER_OF_SENSORS; i++){
            if (sensors_info->updated[i] == FALSE){
                sprintf(cPayload, "%s_: %d_= %d", "puesto", i, sensors_info->state[i]);
                paramsQOS1.payloadLen = strlen(cPayload);

                rc = aws_iot_mqtt_publish(&client, TOPIC, TOPIC_LEN, &paramsQOS1);

                if (rc == SUCCESS){
                    sensors_info->updated[i] = 1;
                }
                else if (rc == MQTT_REQUEST_TIMEOUT_ERROR) {
                    ESP_LOGW(TAG, "QOS1_publish_ack_not_received.");
                    rc = SUCCESS;
                }
                else {
                    ESP_LOGW(TAG, "Problema_al_enviar_el_mensaje");
                }
            }
        }
    }
    else{
        ESP_LOGW(TAG, "sw_aun_no_ha_sido_asignado_por_primera_vez");
    }
}
else{
    ESP_LOGI(TAG, "Esta_iteracion_no_hizo_nada: %d", rc);
}
}

ESP_LOGE(TAG, "An_error_occurred_in_the_main_loop: %d", rc);
abort();
}

void ultrasonic_task(void *pvParameters)
{

```

```

clock_t start, stop;

    ESP_LOGI(pcTaskGetName(0), "Start");

    //Pausa de 10 segundos para esperar a aws-iot-task
    vTaskDelay(10000 / portTICK_PERIOD_MS);

    //Creando arreglo de sensores
    ultrasonic_sensor_t* sensors[NUMBER_OF_SENSORS];

    //Creando arreglo de distancias medidas por los sensores
    uint32_t* distance[NUMBER_OF_SENSORS];

    //Creando estructura para monitorear sensores
    Sensors_info* sensors_info = malloc(sizeof(Sensors_info));

    for (short i = 0; i < NUMBER_OF_SENSORS; i++)
    {
        //Iniciando arreglo de de sensores
        short* sensor_values = map_sensor(i);
        ultrasonic_sensor_t* sensor = malloc(sizeof(ultrasonic_sensor_t));
        sensor->trigger_pin = sensor_values[0];
        sensor->echo_pin = sensor_values[1];
        sensors[i] = sensor;
        if (ultrasonic_init(sensor) != ESP_OK){
            ESP_LOGE(pcTaskGetName(0), "Error inicializando el sensor ultrasonido %d", i);
        }

        ESP_LOGI(pcTaskGetName(0), "Sensor %d: trigger_pin=%d, echo_pin=%d", i, sensors[i]->trigger_pin, sensors[i]->echo_pin);

        //Iniciando arreglo de distancias
        uint32_t distance_init = 0;
        distance[i] = &distance_init;

        //Iniciando estructura sensors_info
        sensors_info->state[i] = INIT;
        sensors_info->updated[i] = 1;
        sensors_info->to_change = 0;
    }

    while (true) {

        for (size_t i = 0; i < NUMBER_OF_SENSORS; i++)
        {
            //Definición e inicialización de variables
            char park_state = DIRTY;
            char sensor_state = sensors_info->state[i];

            start = clock();
            esp_err_t res = ultrasonic_measure_cm(sensors[i], MAX_DISTANCE_CM, distance[i]); //Realizando medición con sensor i

            //Caso: Existe algún problema con el sensor o no se detecta ningún objeto
            if (res != ESP_OK) {
                //printf("Error: ");
                switch (res) {
                    //Error: dispositivo en estado inválido
                    case ESP_ERR_ULTRASONIC_PING:
                        printf("Cannot_ping_(device_is_in_invalid_state)\n");
                        park_state = DIRTY;
                        break;

                    //Error: No se encuentra dispositivo
                    case ESP_ERR_ULTRASONIC_PING_TIMEOUT:
                        printf("Ping_timeout_(no_device_found)\n");
                        park_state = DIRTY;
                        break;

                    //Sensor no detecta ningún objeto
                    case ESP_ERR_ULTRASONIC_ECHO_TIMEOUT:

```

```

        //printf("Echo timeout (i.e. distance too big)\n");
        park_state = FREE; //En este caso se asume que puesto de estacionamiento se encuentra libre
        break;
        default:
            printf("%d\n", res);
    }

}

//Caso: El sensor realiza una medici n exit samente
else {
    //Se analiza el estado del estacionamiento en base a la distancia medida
    float distance_cm = *distance[i];
    park_state = get_park_state(distance_cm);
    //printf("Sensor %d (sensor_state=%d) (park_state=%d) Distance: %"PRIu32" cm.\n", i+1, sensor_state, park_state)

}

//Si cambia estado de estacionamiento i, se actualiza estructura sensors_info y se env a a la cola
if(park_state != sensor_state){
    sensors_info->state[i] = park_state;
    sensors_info->updated[i] = FALSE;
    sensors_info->to_change = i;

    if(xQueueSend(aws_iot_evt_queue, (void*) &sensors_info, portMAX_DELAY )){
        stop = clock();
        ESP_LOGI(TAG, "Mensaje enviado a cola aws_iot: %.3f_s", ((double)(stop - start))/CLOCKS_PER_SEC);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

//
vTaskDelay(100 / portTICK_PERIOD_MS);
}

}

}

static void initialise_wifi(void)
{
    /* Initialize TCP/IP */
    esp_netif_init();

    ESP_ERROR_CHECK(esp_event_loop_create_default());
    ESP_ERROR_CHECK(esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID, &wifi_event_handler, NULL));
    ESP_ERROR_CHECK(esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, &wifi_event_handler, NULL));
    /* Initialize Wi-Fi including netif with default config */
    esp_netif_create_default_wifi_sta();

    wifi_event_group = xEventGroupCreate();
    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK( esp_wifi_init(&cfg) );
    ESP_ERROR_CHECK( esp_wifi_set_storage(WIFI_STORAGE_RAM) );
    wifi_config_t wifi_config = {
        .sta = {
            .ssid = WIFI_SSID,
            .password = WIFI_PASS,
        },
    };
    ESP_LOGI(TAG, "Setting Wi-Fi configuration SSID=%s...", wifi_config.sta.ssid);
    ESP_ERROR_CHECK( esp_wifi_set_mode(WIFI_MODE_STA) );
    ESP_ERROR_CHECK( esp_wifi_set_config(WIFI_IF_STA, &wifi_config) );
    ESP_ERROR_CHECK( esp_wifi_start() );
}

void app_main()
{
    //Creaci n de cola que comunica tareas aws_iot_task y ultrasonic_task

```

```

aws_iot_evt_queue = xQueueCreate(10, sizeof(Sensors.info));

// Si cola no es creada exitosamente abortar
if(aws_iot_evt_queue == NULL){
    ESP_LOGE(TAG,"Error_creando_las_colas\n");
    abort();
}

// Inicializar NVS.
esp_err_t err = nvs_flash_init();
if (err == ESP_ERR_NVS_NO_FREE_PAGES || err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
    ESP_ERROR_CHECK(nvs_flash_erase());
    err = nvs_flash_init();
}
ESP_ERROR_CHECK( err );

//Inicializar wifi
initialise_wifi();

//Crear tarea encargada de mantener la conexi n con AWS y enviar mensajes a este
xTaskCreate(aws_iot_task , "aws_iot_task", 8192 , NULL, 10, NULL);

//Crear tarea encargada de leer datos de los sensores e interpretarlos
xTaskCreate(ultrasonic_task , "ultrasonic_task", 1024*2, NULL, 2, NULL);
}

```

Anexo B

Código fuente de PubSubActivity.java

```
/**
 * Desarrollado por Amazon bajo la licencia "Apache Licence, Version 2.0"
 * Adaptado por Esteban Mohr
 */

package com.amazonaws.demo.androidpubsubwebsocket;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.graphics.Color;
import java.io.UnsupportedEncodingException;
import java.util.UUID;

import com.amazonaws.auth.CognitoCachingCredentialsProvider;
import com.amazonaws.mobileconnectors.iot.AWSIoTClientStatusCallback;
import com.amazonaws.mobileconnectors.iot.AWSIoTManager;
import com.amazonaws.mobileconnectors.iot.AWSIoTNewMessageCallback;
import com.amazonaws.mobileconnectors.iot.AWSIoTQos;
import com.amazonaws.regions.Regions;

public class PubSubActivity extends Activity {

    //Etiqueta utilizada para "logging"
    static final String LOG_TAG = PubSubActivity.class.getCanonicalName();

    // --- Constants to modify per your configuration ---

    // Punto de enlace con AWS. Formato: XXXXXXXXXX.iot.<region>.amazonaws.com,
    private static final String CUSTOMER_SPECIFIC_ENDPOINT = "at0bms8uo1fts-ats.iot.us-east-1.amazonaws.com";

    // ID de grupo de Cognito.
    private static final String COGNITO_POOL_ID = "us-east-1:54f4588a-36a8-41d3-bd81-ab1ffd64b1f0";

    // Regi n de AWS IoT
    private static final Regions MY_REGION = Regions.US_EAST_1;

    TextView tvLastMessage;
    TextView tvClientId;
    TextView tvStatus;
    TextView tvPark1;
    TextView tvPark2;
    TextView tvPark3;
```

```

TextView[] tvParkPlaces;

Button btnConnect;
Button btnSubscribe;
Button btnDisconnect;

AWSIoTmqttManager mqttManager;
String clientId;

CognitoCachingCredentialsProvider credentialsProvider;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    tvLastMessage = (TextView) findViewById(R.id.tvLastMessage);
    tvClientId = (TextView) findViewById(R.id.tvClientId);
    tvStatus = (TextView) findViewById(R.id.tvStatus);
    tvPark1 = (TextView) findViewById(R.id.textViewPuesto1);
    tvPark2 = (TextView) findViewById(R.id.textViewPuesto2);
    tvPark3 = (TextView) findViewById(R.id.textViewPuesto3);

    tvParkPlaces = new TextView[]{tvPark1, tvPark2, tvPark3};

    btnConnect = (Button) findViewById(R.id.btnConnect);
    btnConnect.setOnClickListener(connectClick);
    btnConnect.setEnabled(false);

    btnSubscribe = (Button) findViewById(R.id.btnSubscribe);
    btnSubscribe.setOnClickListener(subscribeClick);

    btnDisconnect = (Button) findViewById(R.id.btnDisconnect);
    btnDisconnect.setOnClickListener(disconnectClick);

    // MQTT client IDs are required to be unique per AWS IoT account.
    // This UUID is "practically unique" but does not _guarantee_
    // uniqueness.
    clientId = UUID.randomUUID().toString();
    tvClientId.setText(clientId);

    // Initialize the AWS Cognito credentials provider
    credentialsProvider = new CognitoCachingCredentialsProvider(
        getApplicationContext(), // context
        COGNITO_POOL_ID, // Identity Pool ID
        MY_REGION // Region
    );

    // MQTT Client
    mqttManager = new AWSIoTmqttManager(clientId, CUSTOMER_SPECIFIC_ENDPOINT);

    // The following block uses a Cognito credentials provider for authentication with AWS IoT.
    new Thread(new Runnable() {
        @Override
        public void run() {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    btnConnect.setEnabled(true);
                }
            });
        }
    }).start();
}

View.OnClickListener connectClick = new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        Log.d(LOG_TAG, "clientId=_ " + clientId);
    }
}

```

```

try {
    mqttManager.connect(credentialsProvider, new AWSToMQTTClientStatusCallback() {
        @Override
        public void onStatusChanged(final AWSToMQTTClientStatus status,
            final Throwable throwable) {
            Log.d(LOG.TAG, "Status_=" + String.valueOf(status));

            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    if (status == AWSToMQTTClientStatus.Connecting) {
                        tvStatus.setText("Connecting ...");
                    } else if (status == AWSToMQTTClientStatus.Connected) {
                        tvStatus.setText("Connected");
                    } else if (status == AWSToMQTTClientStatus.Reconnecting) {
                        if (throwable != null) {
                            Log.e(LOG.TAG, "Connection_error.", throwable);
                        }
                        tvStatus.setText("Reconnecting");
                    } else if (status == AWSToMQTTClientStatus.ConnectionLost) {
                        if (throwable != null) {
                            Log.e(LOG.TAG, "Connection_error.", throwable);
                            throwable.printStackTrace();
                        }
                        tvStatus.setText("Disconnected");
                    } else {
                        tvStatus.setText("Disconnected");
                    }
                }
            });
        }
    });
} catch (final Exception e) {
    Log.e(LOG.TAG, "Connection_error.", e);
    tvStatus.setText("Error!" + e.getMessage());
}
};

View.OnClickListener subscribeClick = new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        //final String topic = txtSubscribe.getText().toString();
        final String topic = "esp32/1";

        Log.d(LOG.TAG, "topic_=" + topic);

        try {
            mqttManager.subscribeToTopic(topic, AWSToMQTTQoS.QOS0,
                new AWSToMQTTNewMessageCallback() {
                    @Override
                    public void onMessageArrived(final String topic, final byte[] data) {
                        runOnUiThread(new Runnable() {
                            @Override
                            public void run() {
                                try {
                                    String message = new String(data, "UTF-8");
                                    String[] splitted_message = message.split(":");
                                    Log.d(LOG.TAG, "Message_arrived:");
                                    Log.d(LOG.TAG, "Topic_=" + topic);
                                    Log.d(LOG.TAG, "Message_=" + message);

                                    tvLastMessage.setText(message);
                                    int parkPlaceNumber = Integer.parseInt(splitted_message[1]) - 1;
                                    int parkPlaceState = Integer.parseInt(splitted_message[2]);
                                    switch (parkPlaceState){

```

```

        case 0:
            tvParkPlaces[parkPlaceNumber].setText("LIBRE");
            tvParkPlaces[parkPlaceNumber].setBackgroundColor(Color.GREEN);
            break;
        case 1:
            tvParkPlaces[parkPlaceNumber].setText("OCUPADO");
            tvParkPlaces[parkPlaceNumber].setBackgroundColor(Color.RED);
            break;
        case 2:
            tvParkPlaces[parkPlaceNumber].setText("REVISAR");
            tvParkPlaces[parkPlaceNumber].setBackgroundColor(Color.YELLOW);
            break;
        default:
            tvParkPlaces[parkPlaceNumber].setText("LIBRE");
            tvParkPlaces[parkPlaceNumber].setBackgroundColor(Color.GREEN);
    }
    //tvParkPlaces[parkPlaceNumber].setText(Integer.toString(parkPlaceState));

    } catch (UnsupportedEncodingException e) {
        Log.e(LOG.TAG, "Message_encoding_error.", e);
    }
    }
    }
    });
    } catch (Exception e) {
        Log.e(LOG.TAG, "Subscription_error.", e);
    }
    }
};

View.OnClickListener disconnectClick = new View.OnClickListener() {
    @Override
    public void onClick(View v) {

        try {
            mqttManager.disconnect();
        } catch (Exception e) {
            Log.e(LOG.TAG, "Disconnect_error.", e);
        }

    }
};
}

```

Anexo C

Explicación detallada del código fuente

Funcionalidad paso a paso de la función *ultrasonic_task* del archivo *smart_parking.c*:

- Realiza una pausa de 10 segundos para esperar a la tarea *aws_iot_task* a ser iniciada.
- Se crea un arreglo de punteros a estructuras *ultrasonic_sensor_t* [43] llamado *sensors*; un arreglo a punteros de enteros llamado *distance* los cuales representan las distancias medidas por los sensores. El tamaño de los dos primeros arreglos es equivalente a la cantidad de sensores. Además se asigna memoria para una instancia de la estructura *Sensors_info* utilizado para monitorear el estado de los sensores.
- Un bucle inicializa los arreglos *sensors* y *distance* como también la estructura *sensors_info*. Al inicializarse el primer arreglo se realiza un mapeo de los índices de este con los respectivos números de pines del microcontrolador en los cuales se conectan los pines *echo* y *trigger* de cada sensor. Este mapeo se realiza con ayuda de la función *map_sensor*.
- Se ejecuta un bucle infinito que realiza lo siguiente para cada uno de los sensores:
 - Se define la variable *park_state* que almacena el estado nuevo del puesto de estacionamiento, luego se realiza una medición con el sensor utilizando la función *ultrasonic_measure_cm* [43].
 - Si el dispositivo se encuentra en estado inválido o no es encontrado se reporta error y se le asigna a la variable *park_state* el estado *Sucio*, en caso que no se detecte ningún objeto se le asigna como valor el estado *libre*.
 - Si el sensor realiza alguna medición de forma exitosa, es decir, detecta algún objeto se le asigna a la variable *park_state* su valor dependiendo de la dis-

tancia medida. Para esto se utiliza la función *get_park_state*, la cual recibe como parámetro la distancia medida en , si esta distancia posee un valor mayor a 50 , retorna el estado **Libre**, si el valor se encuentra entre 10 y 50 retorna el estado y si el valor es menor a 10 retorna el estado *Sucio*.

- Si último estado guardado del sensor (variable *sensor_state*) difiere del estado deducido del puesto de estacionamiento (variable *park_state*) significa que el estado del puesto de estacionamiento ha cambiado, por lo cual debe de ser informado a la tarea *aws_iot_task*. Para realizar esto se envían los valores actuales de la estructura *sensors_info* a través de la cola *aws_iot_evt_queue* haciendo uso de la función *xQueueSend* [44].
- La tarea hace una pausa de 0.1 segundos, esto es necesario para desocupar el procesador y poder ser utilizado por otras tareas.

Funcionalidad paso a paso de la función *aws_iot_task* del archivo *smart_parking.c*:

- Se crea un cliente IoT de AWS (variable *client* de tipo *AWS_IoT_Client*) [45] y la estructura de parámetros de este (variable *mqttInitParams* de tipo *IoT_Client_Init_Params*) [46] a la cual se les asigna sus respectivos valores a los parámetros. Luego se llama a la función *aws_iot_mqtt_init* la cual haciendo uso de las variables creadas inicializa al cliente MQTT.
- Se espera que el WiFi aparezca como conectado, para esto se utiliza la función *xEventGroupWaitBits* [47] la cual, en base a los parámetros utilizados espera que el bit *CONNECTED_BIT* a través de la función *xEventGroupSetBits* [48] presente en la función *wifi_event_handler*.
- Se crea la estructura de parámetros de conexión (variable *connectParams* de tipo *IoT_Client_Connect_Params*) [49] y se les asigna sus respectivos valores a los parámetros. Con esta y la variable *cliente* creada anteriormente se realiza la conexión con AWS con la función *aws_iot_mqtt_connect* [50], si la conexión no es exitosa se reintenta realizar la conexión cada un segundo hasta que esta se lleve a cabo con éxito.
- Se utiliza la función *aws_iot_mqtt_autoreconnect_set_status* [51] para habilitar al cliente de AWS a reconectarse automáticamente en caso que el microcontrolador pierda conexión con sus servicios. Si esta función falla en ejecutarse se aborta el programa.
- Se asigna el tópico de los mensajes a ser enviados a AWS como “esp32/1”.

- Se crea estructura de parámetros de los mensaje publicados (variable *params-QOS1* de tipo *IoT_Publish_Message_Params*) [52] y se asignan sus respectivos valores.
- Se ejecuta un bucle que se mantiene iterando siempre que ningún mensaje falle, es decir, la variable *rc* (“rc” representa “response code”) posee al inicio de cada iteración, los valores *NETWORK_ATTEMPTING_RECONNECT*, *NETWORK_RECONNECTED* o *SUCCESS* [53], en caso contrario quiere decir que ocurre algún error, por lo cual se aborta el programa y se reinicia el microcontrolador. Este bucle realiza los siguientes pasos:
 - Se llama a la función *aws_iot_mqtt_yield* [54] para verificar el estado de conexión del cliente con AWS, esta función está configurada para esperar un máximo de 100 para obtener una respuesta, en caso contrario se considera error de “timeout”.
 - Si la conexión se encuentra en estado *NETWORK_ATTEMPTING_RECONNECT* significa que la conexión se está tratando de restablecer automáticamente, por lo tanto se salta la iteración.
 - En caso que la conexión esté en estado *SUCCESS* se extrae un elemento de la cola *aws_iot_evt_queue* se prepara la carga del mensaje y se publica a AWS. Si este mensaje es enviado exitosamente, se notifica a la estructura *sensors_info* que el estado del puesto de estacionamiento se encuentra actualizado. Si el mensaje no es recibido por error de “timeout” se reintenta tres veces, si este error persiste se aborta el programa. En cualquier otro caso se aborta el programa y es reiniciado el microcontrolador.
 - En caso que la conexión tenga el estado *NETWORK_RECONNECTED* se ejecuta un bucle que actualiza actualiza los estados de todos los puestos de estacionamiento que se encuentren desactualizados, enviando los respectivos mensajes a AWS.

Métodos y Atributos de código fuente de aplicación móvil:

Constantes:

- LOG_TAG: Etiqueta utilizada para registro de la aplicación (“Log”).
- CUSTOMER_SPECIFIC_ENDPOINT: Punto de acceso a AWS.
- COGNITO_POOL_ID: ID del grupo de autenticación de Cognito.

- **MY_REGION:** Región de AWS en la que se transfieren los mensajes.

Atributos:

- *tvLastMessage*: Caja de texto que muestra último mensaje recibido de AWS.
- *tvClientId*: Caja de texto que muestra el identificador del cliente que se conecta a AWS.
- *tvStatus*: Caja de texto que muestra el estado de conexión con la nube de AWS.
- *tvPark1*: Caja de texto que muestra el estado del estacionamiento 1.
- *tvPark2*: Caja de texto que muestra el estado del estacionamiento 2.
- *tvPark3*: Caja de texto que muestra el estado del estacionamiento 3.
- *tvParkPlaces*: Arreglo que contiene referencias a cajas de texto *tvPark1*, *tvPark2* y *tvPark3*.
- *btnConnect*: Botón para conectar aplicación con AWS.
- *btnSubscribe*: Botón para suscribirse al tópico “esp32/1”.
- *btnDisconnect*: Botón para desconectar aplicación de AWS.
- *mqttManager*: Objeto que maneja la conexión con el agente de mensajes de AWS y gestiona las suscripciones y publicaciones de mensajes.
- *clientId*: Identificador del cliente que recibe los mensajes MQTT.
- *credentialsProvider*: Proveedor de credenciales de “Cognito”.

Método *OnCreate*: Este método se activa al crearse la actividad y tiene como funcionalidad asignar el contenido de la vista de la actividad, este encuentra y asigna las vistas a las variables correspondientes. También genera un ID de cliente único mediante el uso de *UUID* [56] y se configura el proveedor de credenciales de AWS Cognito para la autenticación.

Finalmente, habilita el botón de conexión *btnConnect*, la cual se ejecuta en un hilo para evitar bloquear la interfaz de usuario mientras se establece la conexión con AWS IoT.

“Listener” *connectClick*:

Este código define un objeto de tipo *OnClickListener* para el botón “CONNECT”. Cuando el usuario hace clic en el botón, se ejecuta el método *onClick* que contiene la lógica para establecer la conexión MQTT con AWS.

“Listener” *subscribeClick*: Este fragmento de código define un objeto de tipo *OnClickListener* para el botón de “SUBSCRIBE” el cual ejecuta el método *onClick* que

contiene la lógica para suscribirse al tópico “esp32/1” por medio del método *subscribeToTopic*, el tiene tres parámetros: el nombre del tema, el nivel de calidad de servicio (QoS) y un objeto *AWSIoTmqttNewMessageCallback* para manejar los mensajes entrantes y actualizar en la interfaz el estado del correspondiente estacionamiento.

“Listener” disconnectClick:

Este código define un objeto de tipo *OnClickListener* para el botón “DISCONNECT”. Cuando el usuario hace clic en el botón, se ejecuta el método *onClick()* que contiene la lógica para desconectar la conexión MQTT.

Anexo D

Diseño en 3D de carcasa protectora para sensores ultrasonido

Las figuras D.1 y D.2 muestran el modelo en 3D y el plano de vistas de una protección para los sensores infrarrojos diseñada por el Ingeniero en automatización y control industrial Juan Ignacio Ruhe.

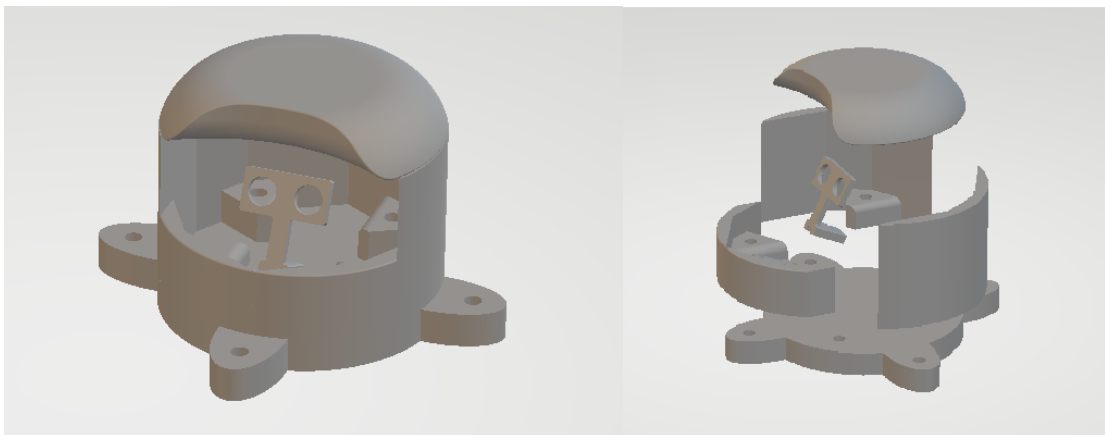


Figura D.1: Modelo 3D de protección para sensores infrarrojos

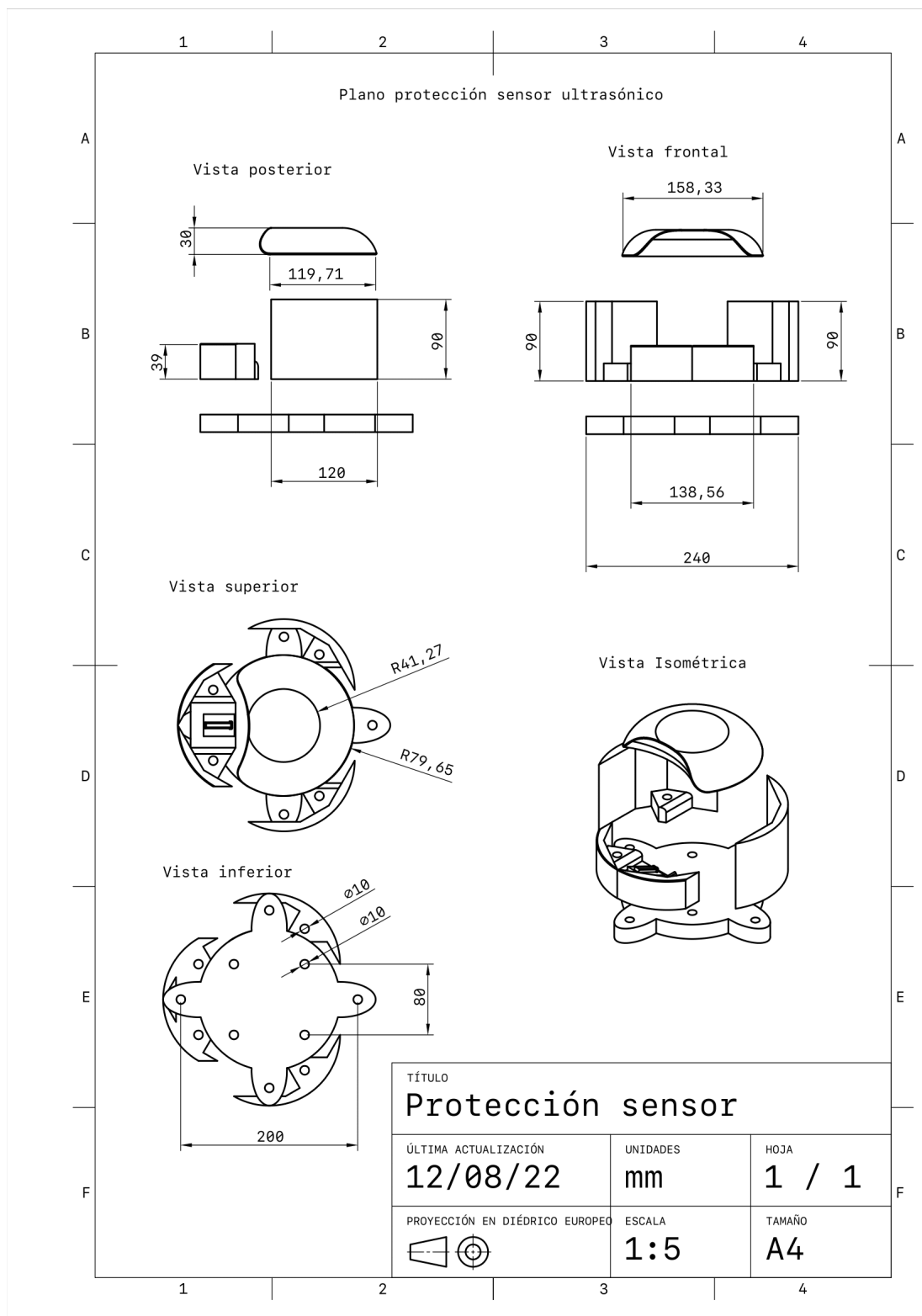


Figura D.2: Plano de vistas de protección para sensores infrarojos