



ugr

Universidad
de Granada

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

TFG - DASIoT

DASIoT: Desarrollo y Auditoría de Seguridad para prototipo de dispositivos IoT

Autor

Luis Aróstegui Ruiz

Directores

José Manuel Soto Hidalgo

Alberto Guillén Perales



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

Granada, 15 de mayo de 2022

TFG - DASIoT: Desarrollo y Auditoría de Seguridad para prototipo de dispositivos IoT

Luis Aróstegui Ruiz

Palabras clave: Internet de las Cosas, Seguridad

Resumen

En el contexto de IoT, hay un ecosistema de entornos de desarrollo específicos. En este TFG se hará uso de alguno de ellos para llevar a cabo la implementación de un dispositivo IoT y analizar los potenciales problemas de seguridad que pueden aparecer durante la etapa de implementación.

Project Title: Project Subtitle

First name, Family name (student)

Keywords: Keyword1, Keyword2, Keyword3,

Abstract

Write here the abstract in English.

Yo, **Nombre Apellido1 Apellido2**, alumno de la titulación **TITULACIÓN de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI **XXXXXXXXXX**, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Nombre Apellido1 Apellido2

Granada a X de mes de 201 .

D. **Nombre Apellido1 Apellido2 (tutor1)**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

D. **Nombre Apellido1 Apellido2 (tutor2)**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado ***Título del proyecto, Subtítulo del proyecto***, ha sido realizado bajo su supervisión por **Nombre Apellido1 Apellido2 (alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

Los directores:

Nombre Apellido1 Apellido2 (tutor1)
(tutor2)

Nombre Apellido1 Apellido2

Agradecimientos

Poner aquí agradecimientos...

Índice general

1. Introducción	17
1.1. Descripción y contexto	17
1.2. Motivación	18
1.3. Objetivos	19
1.3.1. Objetivos específicos	19
2. Estado del arte	21
2.1. Internet de las Cosas	21
2.1.1. Arquitecturas	22
2.1.2. Tecnologías asociadas	24
2.1.3. Aplicaciones	29
2.1.4. Retos	30
2.2. Seguridad	30
2.2.1. Características de seguridad en el IoT	31
2.2.2. Aspectos legales y éticos	31
I Propuesta	33
3. Planificación	35
3.1. Planificación a priori	35
3.1.1. Diagrama de Gantt	35
3.1.2. Etapas de desarrollo	35
3.1.3. Temporización	36
3.1.4. Seguimiento del desarrollo	36
3.2. Planificación a posteriori	38
4. Presupuesto	39

5. Análisis del problema	41
5.1. Middleware	42
5.1.1. Requisitos del servicio de middleware	43
5.1.2. Requisitos de arquitectura en el middleware	44
5.2. Frameworks IoT	45
5.2.1. Requisitos para seleccionar un framework IoT	45
5.2.2. Varios framework IoT	46
5.3. Elección del framework	48
6. Diseño	51
7. Implementación	53
7.1. Conectar dispositivo	54
7.1.1. Términos y conceptos	54
7.1.2. Pasos a seguir	56
7.2. Recogida de datos de un dispositivo	61
7.2.1. Términos y conceptos	62
7.2.2. Pasos a seguir	63
7.3. Envío de comandos al dispositivo	66
7.3.1. Términos y conceptos	66
7.3.2. Pasos a seguir	66
8. Pruebas	71
9. Conclusiones y trabajos futuros	73

Índice de figuras

2.1. Arquitectura de 3 capas. [11]	24
2.2. Arquitectura de 4 capas. [2]	25
3.1. Diagrama de Gantt. Gráfico.	36
5.1. Elementos principales de un framework en IoT. [16]	42
7.1. Dashboard Kaa IoT.	53
7.2. Nombre y descripción de la aplicación	56
7.3. Versiones de la aplicación	57
7.4. Endpoint creado	58
7.5. Datos del dispositivo creado	58
7.6. Habilitar opción de auto extracción de datos	63

Índice de tablas

3.1. Organización temporal del proyecto.	36
5.1. Características de soluciones comerciales para framework IoT. [1]	48

CAPÍTULO 1

Introducción

1.1. Descripción y contexto

El Internet de las Cosas, o IoT, es un sistema de dispositivos informáticos, máquinas mecánicas y digitales, objetos, animales o personas interrelacionados que cuentan con identificadores únicos (UID) y la capacidad de transferir datos a través de una red sin que sea necesaria la interacción entre personas o entre ordenadores. [18]

Una “cosa” en el Internet de las Cosas puede ser una persona con un implante de monitor cardíaco, un animal de granja con un chip, un automóvil que tiene sensores incorporados para alertar al conductor cuando la presión de los neumáticos es baja o cualquier otro objeto natural o artificial al que se le pueda asignar una dirección de Protocolo de Internet (IP) y que sea capaz de transferir datos a través de una red. Cada vez más, las organizaciones de diversos sectores utilizan el IoT para operar de forma más eficiente, comprender mejor a los clientes para ofrecerles un mejor servicio, mejorar la toma de decisiones y aumentar el valor del negocio.

El concepto de *Internet of Things* fue propuesto en 1999 por el laboratorio de identificación automática del Instituto Tecnológico de Massachusetts (MIT). La UIT lo dio a conocer en 2005, empezando por China. El IoT puede definirse como “*datos y dispositivos continuamente disponibles a través de Internet*”. La interconexión de “cosas” (objetos) que pueden dirigirse de forma inequívoca y las redes heterogéneas constituyen el IoT. La identificación por radiofrecuencia (RFID), los sensores, las tecnologías inteligentes y las nanotecnologías son los principales contribuyentes a al IoT para una variedad de servicios. Con la drástica

reducción del coste de sensores y con la evolución de tecnologías como el ancho de banda, el procesamiento, los teléfonos inteligentes, la migración hacia el IPv6 y el 5G se está facilitando la adopción del IoT.

El IoT también ve todo como lo mismo, sin discriminar entre humanos y máquinas. Las “cosas” incluyen a los usuarios finales, los centros de datos (DC), las unidades de procesamiento, los teléfonos inteligentes, las tabletas, el Bluetooth, el ZigBee, la Asociación de Datos por Infrarrojos (IrDA), la banda ultraancha (UWB), las redes celulares, las redes Wi-Fi, los DC de comunicación de campo cercano (NFC), la RFID y sus etiquetas, los sensores y los chips, los equipos domésticos, los relojes de pulsera, los vehículos y las puertas de las casas. [12]

1.2. Motivación

Las personas de todo el mundo están ya preparadas para disfrutar de las ventajas del Internet de las cosas (IoT). El IoT lo incorpora todo, desde el sensor corporal hasta la computación en la nube. Comprende los principales tipos de redes, como la distribuida, la de red, la ubicua y la vehicular, que han conquistado el mundo de la informática durante una década. Desde el estacionamiento de vehículos a su seguimiento, de la introducción de datos de pacientes a la observación de los pacientes a la observación de los pacientes, de la atención a los niños a la atención a los ancianos, de las tarjetas inteligentes a las tarjetas de campo cercano, los sensores están haciendo sentir su presencia. Los sensores desempeñan un papel fundamental en el IoT.

El IoT funciona en redes y estándares heterogéneos. Excepcionalmente, ninguna red está libre de amenazas y vulnerabilidades de seguridad. Cada una de las capas del IoT está expuesta a diferentes tipos de amenazas. Este proyecto se centra en las posibles amenazas que hay que abordar y mitigar para conseguir una comunicación segura en el IoT. [6]

En otras palabras, el IoT combina “lo real y lo virtual” en cualquier lugar y en cualquier momento, atrayendo la atención tanto de desarrolladores como de ciberdelincuentes. Inevitablemente, dejar los dispositivos sin intervención humana durante un largo periodo podría dar lugar a robos. La seguridad ha sido finalmente reconocida como un requisito esencial para todo tipo de sistemas informáticos, incluidos los de IoT. Sin embargo, muchos sistemas IoT son mucho menos seguros que los típicos sistemas Windows/Mac/Linux.

Los problemas de seguridad de IoT se derivan de seguridad de la IO provienen de una serie de causas: características de seguridad inadecuadas en el hardware, software mal diseñado con una serie de vulnerabilidades, contraseñas por defecto

y otros errores de de seguridad. Los nodos IoT inseguros crean problemas para la seguridad de todo el sistema IoT. Dado que los nodos suelen tener una vida útil de varios años, la gran base instalada de de dispositivos inseguros creará problemas de seguridad durante algún tiempo. Los nodos IoT inseguros son ideales para los ataques de denegación de servicio. El ataque Dyn es un ejemplo de ataque basado en el IoT contra la infraestructura tradicional de Internet. Los sistemas IoT inseguros también causan problemas de seguridad al resto de Internet.

La privacidad está relacionada con la seguridad, pero requiere medidas específicas a nivel de aplicación la red y los dispositivos. No sólo hay que proteger los datos de los usuarios contra el robo, sino que la red debe diseñarse de forma que los datos menos privados no puedan utilizarse fácilmente para inferir datos más privados. [18]

1.3. Objetivos

El principal objetivo de este proyecto es hacer uso de un framework específico para llevar a cabo una implementación de un dispositivo IoT y analizar los potenciales problemas de seguridad que pueden aparecer durante la etapa de implementación.

1.3.1. Objetivos específicos

- **OE.1:** Estudiar los distintos frameworks para IoT y analizar funcionalidades y propiedades que se ajusten al proyecto.
- **OE.2:** Desarrollar un prototipo de aplicación para IoT utilizando un framework de desarrollo.
- **OE.3:** Explotación de una vulnerabilidad a nivel de dispositivo, protocolo, SO, aplicación o HW.

CAPÍTULO 2

Estado del arte

Antes de empezar a desarrollar los objetivos del proyecto es necesario conocer primero los fundamentos del Internet de las Cosas, esto engloba desde su funcionamiento, tipos de arquitecturas hasta los distintos estándares que existen.

2.1. Internet de las Cosas

El término “Internet de las Cosas” (IoT) se conoce desde hace unos años. En los últimos tiempos, está recibiendo más atención debido al avance de la tecnología inalámbrica. La idea básica se debe a la variedad de objetos, como RFID, sensores, actuadores, teléfonos móviles, que pueden interactuar entre sí teniendo una dirección distinta. El IoT permite a los objetos ver, oír, pensar y realizar trabajos haciendo que ‘hablen’ entre sí, para compartir y sincronizar información. Transforma estos objetos de convencionales a inteligentes mediante la manipulación de sus tecnologías subyacentes, como los dispositivos integrados, las tecnologías de comunicación, las redes de sensores, los protocolos y las aplicaciones. [19]

La premisa básica y el objetivo de IoT es “conectar lo que no está conectado”. Esto significa que los objetos que no están actualmente unidos a una red informática, es decir, a Internet, se conectarán para que puedan comunicarse e interactuar con personas y otros objetos. IoT es una transición tecnológica en la que los dispositivos nos permitirán sentir y controlar el mundo físico haciendo que los objetos sean más inteligentes y conectándolos a través de una red inteligente. Cuando los objetos y las máquinas pueden ser detectados y controlados a distancia a través de una red, se consigue una mayor integración entre el mundo físico y los ordenadores. Esto permite mejoras en las áreas de eficiencia, preci-

sión, automatización y habilitación de aplicaciones avanzadas.

El mundo del IoT es amplio y puede resultar algo complicado al principio debido a la abundancia de componentes y protocolos que engloba. En lugar de considerar IoT como un único tecnología, es bueno verlo como un paraguas de varios conceptos, protocolos y tecnologías, todos ellos dependientes a veces de un sector concreto. Aunque la amplia gama de elementos de IoT está diseñado para crear numerosos beneficios en las áreas de productividad y automatización, al mismo al mismo tiempo, introduce nuevos retos, como la ampliación del gran número de dispositivos y de la cantidad de datos que deben procesarse. [6]

2.1.1. Arquitecturas

En esta sección se muestran distintas arquitecturas para el IoT, que suele describirse como un proceso de cuatro etapas en el que los datos fluyen desde los sensores conectados a las cosas.^a través de una red y, finalmente, a un centro de datos corporativo o a la nube para su procesamiento, análisis y almacenamiento.

2.1.1.1. Arquitectura de tres capas

Normalmente, la arquitectura de IoT se divide en tres capas básicas: capa de aplicación, capa de red y capa de percepción, que se describen con más detalle a continuación.

- **Capa de percepción.** También conocida como capa de sensores, se implementa como la capa inferior en la arquitectura de IoT. La capa de percepción interactúa con los dispositivos y componentes físicos a través de dispositivos inteligentes (RFID, sensores, actuadores, etc.). Sus principales objetivos son conectar las cosas a la red IoT, y medir, recoger y procesar la información de estado asociada a estas cosas a través de los dispositivos inteligentes desplegados, transmitiendo la información procesada a la capa superior a través de las interfaces de la capa.
- **Capa de red.** También es conocida como la capa de transmisión, se implementa como la capa intermedia en la arquitectura de IoT. La capa de red se utiliza para recibir la información procesada proporcionada por la capa de percepción y determinar las rutas para transmitir los datos y la información al centro del IoT, los dispositivos y las aplicaciones a través de redes integradas. La capa de red es la más importante en la arquitectura de IoT, ya que varios dispositivos (hub, switching, gateway, cloud computing perform, etc.), y varias tecnologías de comunicación (Bluetooth o Wi-Fi) se integran en esta capa. La capa de red debe transmitir datos hacia o desde diferentes cosas o aplicaciones, a través de interfaces o pasarelas entre redes heterogéneas, y utilizando diversas tecnologías y protocolos de comunicación.

- **Capa de aplicación.** También conocida como la capa de negocio, se implementa como la capa superior en la arquitectura de IoT. La capa de aplicación recibe los datos transmitidos desde la capa de red y utiliza los datos para proporcionar los servicios u operaciones requeridos. Por ejemplo, la capa de aplicación puede proporcionar el servicio de almacenamiento para respaldar los datos recibidos en una base de datos, o proporcionar el servicio de análisis para evaluar los datos recibidos para predecir el estado futuro de los dispositivos físicos. Existen varias aplicaciones en esta capa, cada una con requisitos diferentes.

La arquitectura de tres capas es básica para el IoT y se ha diseñado y realizado en varios sistemas. Sin embargo, a pesar de la simplicidad de la arquitectura multicapa de IoT, las funciones y operaciones en las capas de red y aplicación son diversas y complejas. Por ejemplo, la capa de red no sólo necesita determinar rutas y transmitir datos, sino también proporcionar servicios de datos como agregación de datos, computación, etc. La capa de aplicación no sólo necesita proporcionar servicios a los clientes y dispositivos, sino que también debe proporcionar servicios de datos tales como minería de datos, análisis de datos, por ejemplo. Por lo tanto, para establecer una arquitectura multicapa genérica y flexible para la IoT, debe desarrollarse una capa de servicio entre la capa de red y la capa de aplicación para proporcionar los servicios de datos. Basándose en este concepto, recientemente se han desarrollado arquitecturas orientadas a los servicios (SoAs). [13]

2.1.1.2. Arquitectura basada en SoA

En general, SoA (Arquitectura Orientada a Servicios) es un modelo basado en componentes, que puede ser diseñado para conectar diferentes unidades funcionales, también conocidas como servicios, de una aplicación a través de interfaces y protocolos. SoA puede centrarse en el diseño del flujo de trabajo de los servicios coordinados, y permitir la reutilización de los componentes de software y hardware, mejorando la viabilidad de SoA para su uso en el diseño de la arquitectura IoT. Así, SoA puede integrarse fácilmente en la arquitectura de IoT, en la que servicios de datos proporcionados por la capa de red y la capa de aplicación en la arquitectura tradicional de tres capas pueden ser de la arquitectura tradicional de tres capas pueden extraerse y formar una nueva capa, la capa de servicios, también conocida como capa de interfaz o capa de middleware. Así, en una arquitectura de IoT basada en SoA, existen cuatro capas que interactúan entre sí, siendo éstas la capa de percepción, la capa de red, la capa de servicio y la capa de aplicación.

En la arquitectura de IoT basada en cuatro capas, la capa de percepción desempeña la función de la capa inferior de la arquitectura, y se utiliza para medir, recoger y extraer los datos asociados a los dispositivos físicos. La capa de

red se utiliza para determinar las rutas y proporcionar soporte de transmisión de datos a través de redes heterogéneas integradas. La capa de servicio se sitúa entre la capa de red y la capa de aplicación, proporcionando servicios para apoyar la capa de aplicación. La capa de servicio consiste en el descubrimiento de servicios, la composición de servicios, la gestión de servicios y las interfaces de servicios. La capa de aplicación se utiliza para dar soporte a las solicitudes de servicio de los usuarios. [13]

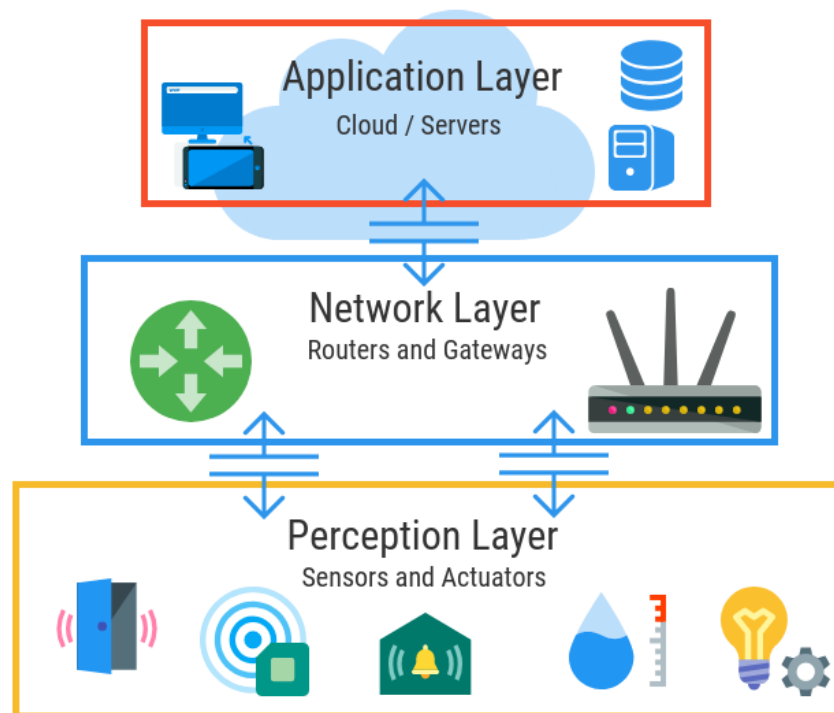


Figura 2.1: Arquitectura de 3 capas. [11]

2.1.2. Tecnologías asociadas

Teniendo en cuenta las arquitecturas mencionadas anteriormente, el IoT cuenta con varias tecnologías que hacen posible la cumplir el objetivo de cada capa de la arquitectura. A continuación se comentan sobre las tecnologías que nos encontramos en la Arquitectura basada en SoA. [13]- [26]

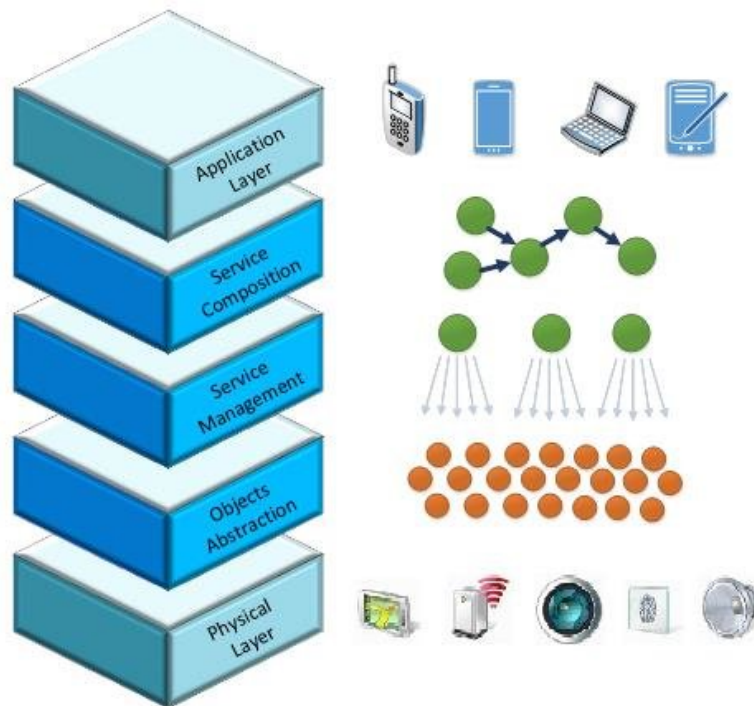


Figura 2.2: Arquitectura de 4 capas. [2]

2.1.2.1. Capa de percepción

En la capa de percepción, la función principal es identificar y rastrear objetos. Para lograr esta función, se pueden implementar las siguientes tecnologías pueden ser implementadas.

- **Radio frequency identification (RFID).** El sistema RFID se compone de uno o varios lectores y varias etiquetas RFID. Utiliza campos electromagnéticos de radiofrecuencia para enviar los datos adjuntos. Las etiquetas que se adjuntan a ella, almacenan datos electrónicamente que pueden ser leídos por RFID cuando se encuentra en la proximidad del lector. [19]
- **Redes de sensores inalámbricos (WSN).** Pueden desempeñar un papel muy importante en el IoT. Las WSN pueden monitorizar y rastrear el estado de los dispositivos, y transmitir los datos de estado al centro de control o a los nodos de enlace a través de múltiples saltos. Por lo tanto, las WSN pueden considerarse como un puente más entre el mundo real y el mundo cibernético. [13]

2.1.2.2. Capa de red

La capa de red se utiliza para determinar el enrutamiento y proporcionar soporte de transmisión de datos a través de redes heterogéneas integradas. A continuación, se presentan algunos protocolos que pueden permitir la comunicación fiable y segura en el IoT. [13]

- **IEEE 802.15.4.** Se lanzó a principios de 2003 y ha satisfecho la necesidad de dispositivos de bajo consumo. Es un protocolo diseñado para la capa física y la capa MAC en redes inalámbricas de área personal (WPAN). El objetivo de IEEE 802.15.4 es centrarse en las WPAN de baja velocidad, proporcionando las conexiones de baja velocidad de todas las cosas en un área personal con bajo consumo de energía, baja tasa de transmisión, y bajo coste. Una de las tecnologías que rompen los esquemas es el estándar de radio IEEE 802.15.4. [10]
- **6LoWPAN.** Las WPAN de baja potencia (LoWPAN) están organizadas por un gran número de dispositivos de bajo coste conectados mediante comunicaciones inalámbricas. En comparación con otros tipos de redes, las LoWPAN presentan una serie de ventajas, como el tamaño reducido de los paquetes, baja potencia, poco ancho de banda, etc. Como mejora, el protocolo 6LoWPAN fue diseñado combinando IPv6 y LoWPAN. En 6LoWPAN, los paquetes IPv6 pueden transmitirse a través de redes IEEE 802.15.4. Debido a que bajo coste y el bajo consumo de energía, 6LoWPAN es adecuado para el IoT, en el que se incluye un gran número de dispositivos de bajo coste.
- **ZigBee.** Es una tecnología de red inalámbrica, diseñada para la comunicación a corto plazo con bajo consumo de energía. En el protocolo ZigBee se incluyen cinco capas la capa física, la capa MAC, la capa de transmisión, la capa de capa de red y la capa de aplicación. Las ventajas de las redes ZigBee son el bajo consumo de energía, el bajo coste baja tasa de datos, baja complejidad, fiabilidad y seguridad.
- **Z-Wave.** Es una tecnología de comunicación inalámbrica a corto plazo con las ventajas de bajo coste, bajo consumo de energía consumo de energía y gran fiabilidad. El objetivo principal de Z-Wave es proporcionar una transmisión fiable entre una control y uno o más dispositivos finales, y Z-wave es adecuada para la red con poco ancho de banda. La red Z-wave soporta la tecnología de enrutamiento dinámico, y cada esclavo almacena una lista en su memoria, que es actualizada por el controlador. La principal diferencia entre ZigBee y Z-wave es la banda de frecuencia en la que opera la capa física.
- **Message Queue Telemetry Transport (MQTT).** Usa la técnica de publicación/suscripción, es un protocolo de mensajería, que se utiliza para

recoger datos medidos en sensores remotos y transmitir los datos a los servidores. MQTT es un protocolo simple y ligero, y soporta la red con bajo ancho de banda y alta latencia. MQTT puede implementarse en varias plataformas para conectar en Internet, y por lo tanto MQTT puede ser utilizado como un protocolo de mensajería entre sensores/actuadores y servidores, lo que hace que MQTT desempeñe un papel importante en el IoT.

- **Constrained Application Protocol (COAP).** Es un protocolo de mensajería basado en la arquitectura de transferencia de estado representativa (REST). Dado que la mayoría de los dispositivos del IoT tienen recursos limitados (es decir, poco almacenamiento y poca capacidad de cálculo), HTTP no puede utilizarse en el IoT, debido a su complejidad. Para superar este problema, se propuso CoAP para modificar algunas funciones de HTTP con el fin de satisfacer los requisitos de el IoT. En términos generales, CoAP es el protocolo de la capa de aplicación en la pila de protocolos 6LoWPAN, y tiene como objetivo permitir que los dispositivos con recursos limitados logren interacciones RESTful. La comunicación de grupo y la notificación push son compatibles con CoAP, pero la difusión no lo es. La observación de recursos, el transporte de recursos en bloque descubrimiento de recursos, interacción con HTTP, y seguridad son todas las características importantes proporcionadas por CoAP.
- **Advanced Message Queuing Protocol (AMQP).** Es un protocolo de colas de mensajes de estándar abierto que se utiliza para proporcionar servicios de mensajes (colas, enrutamiento, seguridad, fiabilidad, etc.) en la capa de aplicación. AMQP se centra en los entornos orientados a mensajes y puede considerarse como un protocolo middleware orientado a mensajes. Usando AMQP, los clientes pueden lograr una comunicación estable con los middlewares de mensajes, incluso si estos clientes y middlewares son producidos por diferentes lenguajes de programación.
- **Extensible Messaging and Presence Protocol (XMPP).** Es un protocolo de mensajería instantánea basado en protocolos de transmisión XML. XMPP hereda las características del protocolo XML, por lo que XMPP tiene una gran escalabilidad, direccionamiento y capacidades de seguridad, y puede ser utilizado para el chat multipartito, voz y vídeo, y telepresencia. En XMPP, se incluyen los siguientes tres funciones: 1) cliente; 2) servidor; y 3) pasarela, así como así como la comunicación bidireccional entre dos partes de estos tres roles.
- **Data Distribution Service (DDS).** Es un protocolo de publicación/suscripción para soportar la comunicación de dispositivo a dispositivo de alto rendimiento. DDS esta centrado en los datos, en el que se puede soportar la multidifusión para lograr una gran QoS y una alta fiabilidad. La ar-

arquitectura de publicación/suscripción sin intermediario hace que DDS sea adecuado para el IoT con restricciones de tiempo real y para la comunicación entre dispositivos. [13]

2.1.2.3. Capa de servicio

Como se ha descrito anteriormente, la capa de servicio se encuentra entre la capa de red y la capa de aplicación, y proporciona servicios eficientes y seguros a los objetos o aplicaciones. En la capa de servicio, deben incluirse las siguientes tecnologías habilitadoras para garantizar que el servicio pueda prestarse de forma eficiente: tecnología de interfaz, tecnología de gestión de servicios, tecnología de middleware y tecnología de gestión y compartición de recursos.

- **Interfaz.** Debe diseñarse en la capa de servicio para garantizar el intercambio de información eficiente y seguro para las comunicaciones entre dispositivos y aplicaciones. Además, la interfaz debe gestionar eficientemente los dispositivos interconectados, incluyendo la conexión de dispositivos, la desconexión de dispositivos, la comunicación de dispositivos y el funcionamiento de dispositivos. Para dar soporte a las aplicaciones en el IoT, un perfil de interfaz (IFP) puede ser considerado como un estándar de servicio, que puede ser utilizado para facilitar las interacciones entre los servicios proporcionados por varios dispositivos o aplicaciones. Para lograr un IFP eficiente, debe implementarse el plug and play universal. Con el desarrollo del IoT, se han realizado varios esfuerzos sobre la interfaz.
- **Gestión de servicios.** Puede descubrir eficazmente los dispositivos y aplicaciones, y programar servicios eficientes y fiables para satisfacer las solicitudes. Un servicio puede considerarse como un comportamiento, que incluye la recogida, el intercambio y el almacenamiento de datos, o una asociación de estos comportamientos para lograr un objetivo especial. En el IoT, algunos requisitos pueden ser satisfechos por un solo servicio, mientras que otros deben ser satisfechos mediante la integración de múltiples servicios.
- **Middleware.** Es un software o servicio de programación que puede proporcionar una abstracción interpuesta entre las tecnologías de IoT y las aplicaciones. En el middleware se ocultan los detalles de las diferentes tecnologías y se proporcionan las interfaces estándar para que los desarrolladores puedan centrarse en el desarrollo de aplicaciones sin tener en cuenta la compatibilidad entre las aplicaciones y las infraestructuras. Así pues, mediante el uso de middleware, los dispositivos y las aplicaciones con diferentes interfaces pueden intercambiar información y compartir recursos entre sí. [13]

2.1.3. Aplicaciones

El IoT es una tecnología emergente que reclama la tercera posición después del ordenador e Internet. En esta tecnología, las cosas se conectan a Internet de una forma u otra, lo que da lugar a una enorme cantidad de datos que deben ser procesados, almacenados y representados de forma eficiente. Las aplicaciones de el IoT varían en diferentes campos e incluyen aplicaciones personales, industriales y nacionales. En los últimos años, estas aplicaciones han aumentado constantemente y algunas de ellas ya están desplegadas y se utilizan a diferentes niveles. Las aplicaciones de IoT requieren hardware, middleware y presentación. Estas aplicaciones tienen características como la comunicación de dispositivos con el mundo real, interacción con el entorno interacción entre personas y dispositivos, tareas rutinarias automáticas con menos supervisión, infraestructura organizada y seguridad en la comunicación.

Son muchas las aplicaciones que se están desarrollando hoy en día y en los últimos años las aplicaciones de IoT han ido a más. Las aplicaciones requieren un avance RFID y tecnologías de direccionamiento, con mejor visualización y capacidad de almacenamiento. Teniendo en cuenta los requisitos de IoT se ha desarrollado una arquitectura de aplicaciones para un menor consumo de energía. Algunos ejemplos que nos encontramos de aplicaciones del IoT son: [10]- [26]

- **Casas inteligentes.** En este caso nos encontramos nuevos inventos que se conectan y controlan nuestros hogares desde nuestros teléfonos.
- **Wearable Technology.** Se han diseñado e implementado varios productos diseñados y aplicados en este ámbito, desde anillos a calzado, y desde relojes hasta mochilas. Esta tecnología puede verse en todas partes.
- **Ciudades inteligentes.** Es un área de tecnología al servicio de las personas, y esta tecnología se construye esencialmente en torno al usuario. Una ciudad no es más que una red diseñada para optimizar los recursos.
- **Red inteligente.** Desde que las instalaciones del IoT se han añadido a la red anterior de datos, se han vuelto más inteligentes, lo que significa más y más información de la red. Desde que el IoT ha ayudado a convertirse en el sistema de suministro de energía de dos maneras la información se enviará al centro principal sobre el uso y el consumo.
- **Atención sanitaria inteligente.** Es un elemento vital para nuestra salud y bienestar, ya sea para el hospital, la odontología o la residencia de ancianos. Este apunta a todos los grupos de edad de la población al mismo tiempo. Para ello se requiere eficacia y menos costes.

2.1.4. Retos

Muchas empresas se apresuran a crear aplicaciones para el IoT y gastan enormes sumas de dinero porque es la próxima gran oportunidad. Se han introducido lavadoras, sistemas de calefacción y frigoríficos inteligentes. El IoT ha traído muchos cambios y dimensiones positivas. Sin embargo, tienen una serie de riesgos y desafíos. [10]- [26]

- **Seguridad y privacidad.** Las aplicaciones del IoT y las áreas inteligentes manejan muchos datos a diario y estos datos se comparten entre dispositivos. La información sobre casas, edificios y coches se comparte entre dispositivos todo el tiempo. Toda esta información exige un mejor sistema de gestión. Los sistemas de seguridad existentes no son adecuados en muchos aspectos y pueden causar graves problemas al usuario. Un error puede hacer que la seguridad del hogar, los coches, los edificios y los datos guardados sean vulnerables.
- **Comunicación fiable.** El IoT se compone de tecnologías fijas y móviles, y se intenta conseguir una comunicación bidireccional sin pérdida de datos y con total fiabilidad.
- **Consumo de energía.** Los dispositivos deben ser capaces de comunicarse con un menor uso de la batería porque estos dispositivos se despliegan lejos y necesitan funcionar con baterías.
- **Interoperabilidad.** Este requiere que los dispositivos se comuniquen entre sí. A nivel de usuario, el usuario puede tener múltiples dispositivos que funcionen en múltiples plataformas. El usuario no querrá quedarse con un solo dispositivo. En cambio, un usuario puede querer diversidad en caso de elegir un dispositivo.

2.2. Seguridad

Se prevé que el Internet de las cosas crezca rápidamente debido a la proliferación de la tecnología de la comunicación, la disponibilidad de los dispositivos y los sistemas informáticos. Por lo tanto, la seguridad de IoT es un área de preocupación para proteger el hardware y las redes en el sistema del IoT. El objetivo principal de la seguridad de la IoT es preservar la privacidad, la confidencialidad, garantizar la seguridad de los usuarios, las infraestructuras, los datos y los dispositivos de la IoT, y garantizar la disponibilidad de los servicios ofrecidos por un ecosistema de IoT. Por lo tanto, la investigación sobre la seguridad del IoT ha cobrado recientemente un gran impulso con la ayuda de las herramientas de simulación, los modeladores y las plataformas de cálculo y análisis disponibles. En la actualidad, las técnicas y los métodos de seguridad que se han propuesto se basan esencialmente en los métodos de seguridad de red convencionales. Sin

embargo, la aplicación de mecanismos de seguridad en un sistema IoT es más difícil que en una red tradicional, debido a la heterogeneidad de los dispositivos y protocolos, así como a la escala o el número de nodos del sistema. [13]- [7].

2.2.1. Características de seguridad en el IoT

- **Confidencialidad.** La confidencialidad puede garantizar que los datos sólo estén disponibles para los usuarios autorizados durante todo el proceso, y que no puedan ser escuchados o interferidos por usuarios no autorizados. En IoT, la confidencialidad es un principio de seguridad importante, ya que un gran número de dispositivos de medición (RFID, sensores, etc.) pueden integrarse en IoT. Por lo tanto, es fundamental garantizar que los datos recogidos por un dispositivo de medición no revelen información segura a sus dispositivos vecinos. Para lograr una gran confidencialidad, deben mejorarse las técnicas, incluyendo mecanismos seguros de gestión de claves.
- **Integridad.** La integridad puede garantizar que los datos no puedan ser manipulados por interferencias intencionadas o no intencionadas durante la entrega de datos en las redes de comunicación, proporcionando en última instancia los datos precisos para los usuarios autorizados. La integridad es importante para IoT, porque si las aplicaciones de IoT reciben datos falsificados o manipulados, se puede estimar un estado de funcionamiento erróneo y se pueden realizar comandos de retroalimentación equivocados, lo que podría interrumpir aún más el funcionamiento de las aplicaciones de IoT. Para lograr una integridad aceptable, deben desarrollarse y aplicarse mecanismos de integridad de datos seguros mejorados y aplicarlos.
- **Disponibilidad.** La disponibilidad puede garantizar que los datos y los dispositivos estén disponibles para los usuarios y servicios autorizados siempre que se soliciten los datos y los dispositivos. En IoT, los servicios suelen solicitarse en tiempo real, y los servicios no pueden programarse y prestarse si los datos solicitados no pueden entregarse a tiempo. Por lo tanto, la disponibilidad es también un importante principio de seguridad. Una de las amenazas más graves para la disponibilidad es el ataque de denegación de servicio (DoS), y las técnicas mejoradas, protocolos de enrutamiento seguros y eficientes, para garantizar la disponibilidad en el IoT.

2.2.2. Aspectos legales y éticos

I | Propuesta

CAPÍTULO 3

Planificación

3.1. Planificación a priori

Se trata de planificar como se espera desarrollar el proyecto en el tiempo, para ello se va a hacer uso de un diagrama de Gantt donde se va a proporcionar una vista general de las tareas programadas, estas tareas tendrán que completarse en unas fechas estipuladas.

3.1.1. Diagrama de Gantt

Un diagrama de Gantt es una herramienta útil para planificar proyectos. Proporciona una vista general de las tareas programadas, indicando el periodo de tiempo que tienen para completarse.

El diagrama se mostrará:

- Fecha de inicio y finalización del proyecto.
- Las tareas del proyecto.
- Fecha de programada de cada tarea, tanto la de inicio como la de final.
- Como se superponen las tareas y si hay relación entre ellas.

Con esta planificación se conseguirá una mayor claridad en las tareas a realizar y una mejor gestión del tiempo.

3.1.2. Etapas de desarrollo

- **1ª etapa:** Revisar frameworks para IoT.

- 2ª etapa: Desarrollar aplicación para IoT.
- 3ª etapa: Explotación de una vulnerabilidad.
- 4ª etapa: Documentación.

3.1.3. Temporización

De los 3 meses y medios a que se van a dedicar al proyecto, en todos ellos se va a desarrollar las etapas indicadas anteriormente. Se muestra la fecha de inicio y de fin de cada etapa.

Etapas del desarrollo	Fecha de comienzo	Fecha de finalización
Documentación del proyecto	9 de Marzo	23 de Junio
Revisar frameworks IoT	21 de Marzo	6 de Abril
Desarrollar aplicación para IoT	7 de Abril	12 de Mayo
Explotar vulnerabilidad	13 de Mayo	22 de Junio

Tabla 3.1: Organización temporal del proyecto.

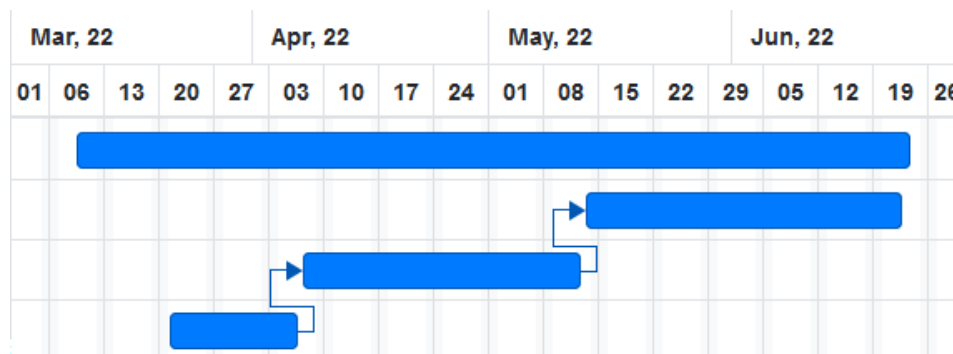


Figura 3.1: Diagrama de Gantt. Gráfico.

Este gráfico corresponde con la duración de cada etapa, comenzando con la documentación. También se indica que hay etapas que para que estas se puedan empezar a desarrollar necesitan que la anterior este terminada, es el caso de la etapa de *Desarrollo de la aplicación* y de *Explotación de una vulnerabilidad*.

3.1.4. Seguimiento del desarrollo

Con el fin de facilitar la visualización del progreso del proyecto se usan distintas herramientas.

3.1.4.1. Trello

Permite gestionar proyectos de forma sencilla y muy visual. Por cada proyecto se crea un tablero donde podemos crear listas y cada lista almacenará *tarjetas* donde se describirá una tarea u objetivo a cumplir. En nuestro caso el tablero se divide en 7 listas:

- **Dudas**, se dejan ancladas preguntas a los tutores.
- **Objetivos**, para recordar los objetivos que se tienen que completar para el proyecto.
- **Pendientes**, tareas que se tienen que realizar pero no se están desarrollando.
- **En proceso**, tareas que se están trabajando.
- **Pendientes de revisión**, tareas finalizadas que requieren de revisión por parte de los tutores.
- **Terminadas**, tareas que han sido aceptadas en la revisión.
- **Hitos**, se agrupan tareas que forman parte de una misma etapa.

3.1.4.2. Github

Se ha creado un repositorio ¹ donde se van añadiendo *issues* por cada tarea que se tenga que realizar. Esta tarea es la misma que nos encontramos en el tablero de Trello, por tanto, cuando se crea una nueva tarea en Trello, creamos un nuevo issue con el mismo nombre y descripción para seguir un progreso coherente.

También se crean *Milestones* que se corresponde con el nombre y descripción de la lista de *Hitos* del tablero de Trello.

Cada vez que terminemos de trabajar en un *Milestone* se creará un *Pull Request* para que los tutores puedan revisar los cambios del proyecto y aceptarlos o rechazarlos. Cada acción que se realice tanto en Trello como en Github se verá reflejado en ambos, de esta manera conseguimos un desarrollo del proyecto realista de principio a final.

3.1.4.3. Clockify

Esta herramienta nos permite seguir el tiempo que estamos trabajando. Cada vez que nos pongamos a trabajar, iniciaremos el contador y nombraremos a ese contador con el nombre de la tarea que estemos realizando en ese momento.

¹Enlace al repositorio: <https://github.com/LuisArostegui/TFG>

3.2. Planificación a posteriori

CAPÍTULO 4

Presupuesto

CAPÍTULO 5

Análisis del problema

Siguiendo los objetivos del proyecto 1.3 se va a analizar los diferentes frameworks que existen para desarrollar una aplicación para posteriormente analizar posibles explotaciones de seguridad.

Como hemos visto en 2.1, normalmente, cuando se generan grandes datos y se transmiten a través de varios dispositivos, tiene que haber un punto específico en el que se recoja y combine todo. Este punto específico es muy esencial en una red, ya que combina todos los datos, lo que permite comprender los datos que se generan. Sin embargo, la transmisión y generación de datos sin problemas no se produce sin más. Más bien, suele ser posible gracias a un framework del Internet de las Cosas.

Un framework para IoT puede describirse como un ecosistema, compuesto por varios dispositivos conectados que se comunican entre sí, a través de Internet. Estos dispositivos conectados suelen funcionar para transferir y detectar datos a través de Internet, y requieren muy poca intervención humana. El framework de IoT es lo que hace posible que los dispositivos conectados tengan una comunicación fluida a través de Internet. Es un elemento tecnológico muy importante en el mundo moderno, que encuentra aplicación en casi todos los sectores. Por ejemplo, una de las principales aplicaciones del IoT es el diseño de casas inteligentes. [9]

Es decir, un framework para IoT recoge todos principales elementos que componen el mundo del Internet de las Cosas para desarrollar una aplicación, estos elementos los podemos ver en la siguiente imagen 5.1.

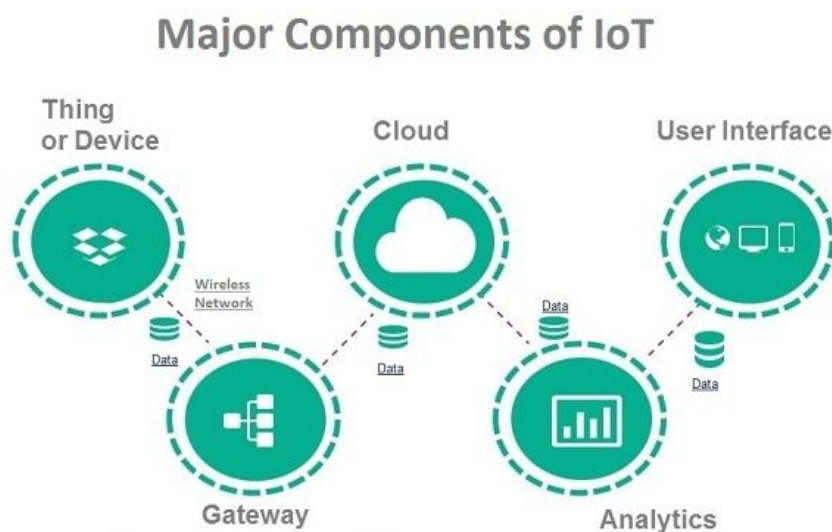


Figura 5.1: Elementos principales de un framework en IoT. [16]

Cuando se esta desarrollando una aplicación, el concepto de “framework” se puede relacionar con otros conceptos como el de “middleware” y “plataforma”. El concepto de “plataforma” hace referencia al lugar donde se permite desplegar y ejecutar la aplicación. Por otro lado el middleware, tiene que ver con la prestación de los servicios. Y por último, el framework se centra más en el diseño del software. Por esto, pese a que en este proyecto se va a trabajar con un framework, hay que tener en cuenta otros conceptos que rodean al desarrollo de una aplicación. [14]

5.1. Middleware

Generalmente, un middleware abstrae las complejidades del sistema o del hardware, permitiendo al desarrollador de la aplicación centrar todo su esfuerzo en la tarea a resolver, sin la distracción de preocupaciones a nivel del sistema o del hardware. Dichas complejidades pueden estar relacionadas con problemas de comunicación o de computación más generales. Un middleware proporciona una capa de software entre las aplicaciones, el sistema operativo y las capas de comunicación de la red, que facilita y coordina que facilita y coordina algún aspecto del procesamiento cooperativo. Desde el punto de vista perspectiva informática, un middleware proporciona una capa entre el software de aplicación y el software de sistema.

En el IoT, es probable que haya una considerable heterogeneidad tanto en las tecnologías de comunicación en uso, como en las tecnologías a nivel de sistema,

y un middleware debe soportar ambas perspectivas según sea necesario.

Basándonos en las características descritas anteriormente 2.1 de la infraestructura del IoT y de las aplicaciones que dependen de ella, se establecen un conjunto de requisitos para que un middleware soporte el IoT. A continuación, estos requisitos se agrupan en dos conjuntos: los servicios que debe proporcionar dicho middleware y la arquitectura del sistema [17]- [15].

5.1.1. Requisitos del servicio de middleware

Los requisitos de los servicios de middleware para el IoT pueden clasificarse en funcionales y no funcionales. Los requisitos funcionales recogen los servicios o funciones, como por ejemplo abstracciones y gestión de recursos un middleware, y los requisitos no funcionales, por ejemplo fiabilidad, seguridad y disponibilidad, captan el soporte de la calidad de servicio o del rendimiento.

En los requisitos funcionales nos encontramos:

- **Descubrimiento de recursos.** Ya que la infraestructura y el entorno del IoT son dinámicos, las suposiciones relacionadas con el conocimiento global y determinista de la disponibilidad de estos recursos es inviable. Pasa lo mismo con la intervención humana, es inviable que este descubra recursos. Por tanto, es importante destacar que el descubrimiento de recursos este automatizado. Cuando no hay infraestructura, el propio dispositivo debe de anunciar su presencia y los recursos que ofrece.
- **Gestión de recursos.** Se espera una QoS (Quality of Service) aceptable para todas las aplicaciones, y en un entorno en el que los recursos que influyen en la QoS son limitados, como el IoT, es importante que las aplicaciones cuenten con un servicio que gestione esos recursos.
- **Gestión de datos.** Los datos son fundamentales en las aplicaciones de IoT. En el IoT, los datos se refieren principalmente a los datos detectados o a cualquier información de infraestructura de red de interés para las aplicaciones.
- **Gestión de eventos.** En las aplicaciones de IoT se genera potencialmente un número masivo de eventos, que deberían gestionarse como parte integral de un middleware de IoT.
- **Gestión del código.** El despliegue de código en un entorno de IoT es un reto, y debe ser soportado directamente por el middleware. En particular, se necesitan servicios de asignación y de código.

En los requisitos no funcionales nos encontramos:

- **Escalabilidad.** Un middleware de IoT debe ser escalable para adaptarse al crecimiento de la red y las aplicaciones/servicios de IoT.

- **Actuación en tiempo real.** Un middleware debe proporcionar servicios en tiempo real cuando la corrección de una operación que soporta depende no sólo de su corrección lógica sino también del tiempo en que se realiza.
- **Fiabilidad.** Un middleware debe permanecer operativo durante la duración de una misión, incluso en presencia de fallos.
- **Disponibilidad.** Un middleware que soporte las aplicaciones de un IoT, especialmente las de misión crítica, debe estar disponible en todo momento.
- **Seguridad y privacidad.** La seguridad es fundamental para el funcionamiento de IoT. En el middleware de IoT, la seguridad debe tenerse en cuenta en todos los bloques funcionales y no funcionales, incluyendo la aplicación a nivel de usuario.
- **Facilidad de despliegue.** Hay que evitar los procedimientos complicados de instalación y configuración.
- **Popularidad.** Un middleware de IoT, como cualquier otra solución de software, debe recibir un apoyo y una ampliación continua.

5.1.2. Requisitos de arquitectura en el middleware

En esta sección se muestran los requisitos para las abstracciones de programación y otros aspectos relacionados con la implementación.

- **Abstracción de la programación.** Para el desarrollador de aplicaciones o servicios, las interfaces de programación de alto nivel deben aislar el desarrollo de las aplicaciones o los servicios de las operaciones proporcionadas por las infraestructuras subyacentes y heterogéneas del IoT.
- **Interoperabilidad.** Un middleware debe funcionar con dispositivos, tecnologías y aplicaciones heterogéneas, sin esfuerzo adicional por parte del desarrollador de aplicaciones o servicios.
- **Basado en el servicio.** Una arquitectura de middleware debe estar basada en servicios para ofrecer una gran flexibilidad cuando sea necesario añadir funciones nuevas y avanzadas al middleware de un IoT.
- **Adaptable.** Un middleware debe ser adaptativo para poder evolucionar y adaptarse a los cambios de su entorno.
- **Consciente del contexto.** El conocimiento del contexto es un requisito clave en la construcción de sistemas adaptativos y también en el establecimiento de valor de los datos recogidos.

- **Autónomo.** Los dispositivos, tecnologías y aplicaciones son participantes activos en los procesos del IoT y deben estar habilitados para interactuar y comunicarse entre sí sin la intervención humana.
- **Distribuido.** Las aplicaciones de un sistema IoT a gran escala de un sistema IoT intercambian información y colaboran entre sí.

5.2. Frameworks IoT

Una vez hemos visto requisitos que necesitamos en nuestro middleware, se van a comentar una serie de requisitos que buscamos en nuestro framework IoT. [1]- [4]- [14]

5.2.1. Requisitos para seleccionar un framework IoT

Algunos requisitos principales en los que nos tenemos que fijar a la hora de seleccionar un entorno de desarrollo son:

- **Seguridad y privacidad.** El framework debe de proporcionar seguridad en la capa de transporte para conexiones seguras (encriptación SSL) y no comprometer en el entorno de trabajo. Con esto, conseguimos privacidad, autenticación e integridad en los datos.
- **Dificultad de uso.** Seleccionar un framework que sea sencillo de usar para desarrolladores. Esto incluye el lenguaje de programación que se use para hacer la aplicación.
- **Código abierto.** Se trata de evitar aquellos framework en los que hay que pagar por su uso o que den un periodo de prueba gratuito. Las plataformas de código abierto permiten que las plataformas sean gestionadas por el desarrollador según sus necesidades.
- **Soporte.** Se busca un framework que sea popular y tenga un equipo y comunidad activa, que continuamente este aportando nuevas características al proyecto.

También existen otras características en las que nos podemos fijar que pueden resultar útiles dependiendo del tipo de aplicación que queremos desarrollar:

- **Tipo de soporte de protocolo de comunicación.** Los dispositivos IoT soportan múltiples protocolos de comunicación. Algunos son ligeros y otros son seguros. El protocolo a elegir depende de los requisitos de la aplicación IoT. CoAP es similar a HTTP pero es ligero, por lo que es más adecuado para aplicaciones móviles. MQTT también es ligero y soporta el concepto de broker, por lo que es bueno para aplicaciones de ancho de banda limitado. CoAP es bueno para la multidifusión y la difusión.

- **Disponibilidad.** La disponibilidad y la estabilidad son parámetros importantes para los requisitos del IoT. Por ejemplo, una aplicación enfocada a la salud requiere que los datos médicos del paciente para ser monitorizados continuamente.
- **Tecnologías de almacenamiento utilizadas.** Diferentes tecnologías de almacenamiento y procesamiento sobre la nube soportan diferentes tipos de análisis. Según los requisitos de procesamiento de datos, se puede seleccionar una nube con diferentes tecnologías de almacenamiento.
- **Tipo de análisis soportado.** Las aplicaciones de IoT suelen requerir datos en tiempo real o datos históricos para el desarrollo de la aplicación. La elección de la tecnología adecuada para aplicar la analítica al tipo de datos generados por el dispositivo en la aplicación es otro factor importante.

5.2.2. Varios framework IoT

5.2.2.1. KAA IoT

Es totalmente gratis, tiene capacidad para gestionar millones de sensores, recoger y analizar datos en tiempo real y visualizarlos, gestionar y conectar productos inteligentes con ayuda de nube. Permite la gestión de los datos de los objetos conectados y la infraestructura de back-end que proporciona los componentes del SDK del servidor y del endpoint. KAA proporciona la funcionalidad de back-end necesaria para operar una solución IoT. Proporciona flexibilidad a los usuarios para implementar sus propias políticas de seguridad.

Kaa proporciona un gateway que aporta la posibilidad de conectar diferentes redes entre si. Los protocolos de comunicación que usa para comunicarse con los dispositivos son MQTT y COAP. Para especificar un dispositivo se hace en JSON, donde cada dispositivo es un objeto. Kaa permite el análisis de datos y para ello usa tecnologías como NoSQL, Cassandra, Hadoop y MongoDB. Kaa soporta lenguajes de programación como Java, C, C++. [21]

5.2.2.2. ThingSpeak

Es un framework que se caracteriza por sus visualizaciones y predicciones utilizando MATLAB. Soporta datos con formato JSON y XML. Es de código abierto. Soporta múltiples dispositivos y permite utilizar protocolos como MQTT o Rest API para la comunicación entre dispositivos. Proporciona seguridad en la capa de transporta con encriptación en las comunicaciones. [25]

5.2.2.3. Macchina.io

Este framework se divide en dos productos, el primero es el **Edge** que permite que las aplicaciones se ejecuten en dispositivos basados en Linux utilizando

C++ y Javascript. Y el segundo producto es el **Remote** que permite gestionar la infraestructura mediante paneles web y aplicaciones móviles. Una de las tecnologías que utiliza es SQLite e incluye soporte para protocolos de comunicación como MQTT, SOAP, HTTP. [5]

5.2.2.4. Altair (Carriots)

Altair, anteriormente "Carriots", es una plataforma diseñada para proyectos IoT. Permite integrar los dispositivos de IoT a una aplicación externa que requiera de los datos mientras ellos se encargan del almacenamiento y la comunicación. Soporta XML, JSON y API Rest. No es gratis, tiene un plan de suscripción. Permite escribir aplicaciones en Java. Para la comunicación entre dispositivos usa MQTT, no posee encriptación en la comunicación pero si posee un mecanismo de autenticación y autorización. Para el almacenamiento usa NoSQL. [8]

5.2.2.5. Zetta

Es una plataforma de código abierto construida sobre Node.js para crear servidores del Internet de las Cosas que se ejecutan a través de ordenadores geodistribuidos y la nube. Zetta combina las API de REST, los WebSockets y la programación reactiva, lo que resulta perfecto para ensamblar muchos dispositivos en aplicaciones de uso intensivo de datos en tiempo real. Zetta tiene la capacidad de convertir cualquier dispositivo en una API. Al comunicarse con microcontroladores como Arduino y Spark Core, Zetta puede proporcionar a cada dispositivo una API REST tanto localmente como en la nube. Zetta es "developer friendly" lo que significa que es sencillo desarrollar usando esta plataforma. Entre las tecnologías que usa esta API Rest y JSON. [3]

5.2.2.6. Temboo

Es un framework que entra dentro de la categoría de plataformas que son "developer friendly", es decir, para conectar dispositivos y hacer tareas simples se hace de manera rápida y sin complicaciones en el código. Soporta datos en Excel, CSV, XML y JSON. Soporta lenguajes de programación como C, Java, Python y Javascript. No es gratuito, requiere de un plan de suscripción para empezar a utilizarlo. Tiene soporte con múltiples dispositivos, entre ellos Arduino y a destacar Samsung Artik. Los protocolos que soporta son HTTP, MQTT, CoAP. Y tiene soporte de tecnologías como Microsoft Power BI y Google BigQuery. [24]

5.2.2.7. Particle

Particle es una plataforma de IoT escalable, fiable y segura. Una característica clave es la capacidad de ejecutar código de cableado de Arduino y Raspberry Pi haciendo más fácil conectar componentes electrónicos a la nube. Los primeros 100 dispositivos que se conecten son gratuitos. En cuanto a la seguridad, es uno

de los frameworks con más características relacionadas con esta, posee encriptación en las comunicaciones, autenticación, autorización y posibilidad de auditoría de seguridad, todo esto bajo el protocolo HTTP. Soporta datos en formato CSV y se puede desarrollar código Javascript y en particle js, una librería ligera de Javascript.

5.2.2.8. Soluciones comerciales

Hay otras soluciones comerciales donde nos encontramos plataformas como AWS IoT, Microsoft Azure IoT Hub, IBM Watson IoT, Google IoT y Oracle IoT. Todas estas alternativas son bastante similares a todas las comentadas hasta el momento, se diferencian entre si por ciertas características como el soporte en lenguajes de programación, donde aparecen lenguajes que no vemos en otras plataformas como son Ruby, .NET, Go, php. En la siguiente tabla se muestran características de estas plataformas :

Framework IoT	Soporte de datos	Soporte de lenguajes de programación	Protocolos	Seguridad
AWS IoT	JSON	Java, C, NodeJS, Javascript, Python, iOS, Android	HTTP, MQTT, Websockets	Encriptación, autenticación, autorización, auditoría
Microsoft Azure IoT	JSON	.NET, UWP, Java, C, NodeJS, Ruby, Android, iOS	HTTP, AMQP	Encriptación, autenticación, autorización, políticas definidas por usuario
IBM Watson IoT	JSON, CSV	C#, C, Python, Java, NodeJS	MQTT	Autenticación, Autorización
Oracle IoT	JSON, CSV	Java, Javascript, Android, C, iOS	REST API	Autenticación, Autorización
Google IoT	JSON	Go, Java, .NET, Node.js, php, Python, Ruby	MQTT, HTTP	Autenticación

Tabla 5.1: Características de soluciones comerciales para framework IoT. [1]

5.3. Elección del framework

Como framework para desarrollar la aplicación se ha optado por trabajar con **Kaa IoT**. La motivación de su uso viene de cumplir todos los requisitos establecidos en la sección 5.2.1.

- **Seguridad y privacidad.** Kaa IoT proporciona autenticación del cliente

con certificado SSL/TLS, por tanto, todas sus comunicaciones van a estar cifradas. Aparte, para el proyecto puede resultar útil posible implementación de nuestras propias políticas de seguridad.

- **Dificultad de uso.** No llega a ser un framework “developer friendly”, pero en la documentación del framework [21] hay varios tutoriales como para desarrollar sin mucha dificultad una aplicación, explicando en cada apartado las palabras clave o conceptos que se necesitan para entender todo lo que se esta haciendo en la construcción de la aplicación. Aquellos frameworks “developer friendly”, por el hecho de ser plataformas donde ofrecen ayudar por desarrollar y desplegar la aplicación requieren de una suscripción para hacer uso de estas.
- **Código abierto.** Es un framework gratuito, evitamos todos los frameworks que requieren de un plan de suscripción.
- **Soporte.** Es de los frameworks más conocidos. Y tienen una comunidad activa, esto se puede observar en su repositorio de github ¹, que cada poco tiempo van añadiendo nuevas características y documentación al proyecto.

Otra elección interesante podría haber sido **macchina.io** no cumple con el requisito de dificultad de uso. En comparación con **Kaa IoT**, macchina.io no dispone de tanta documentación como el framework seleccionado.

¹Enlace a repositorio de Kaa IoT: <https://github.com/kaaproject/kaa>

CAPÍTULO 6

Diseño

CAPÍTULO 7

Implementación

La implementación de la aplicación se realiza con **Kaa IoT** tal y como se analizó anteriormente 5.3. Para empezar a usar el framework, hay que registrarse en sistema. Una vez registrados tendremos acceso a nuestro *dashboard* ¹. En este capítulo se trata de mostrar una guía con la que se consiga conectar un dispositivo, recoger y enviar datos desde/hacia el dispositivo y mostrar las opciones que nos ofrece Kaa IoT como framework IoT. Para empezar, vamos a conectar nuestro primer dispositivo.

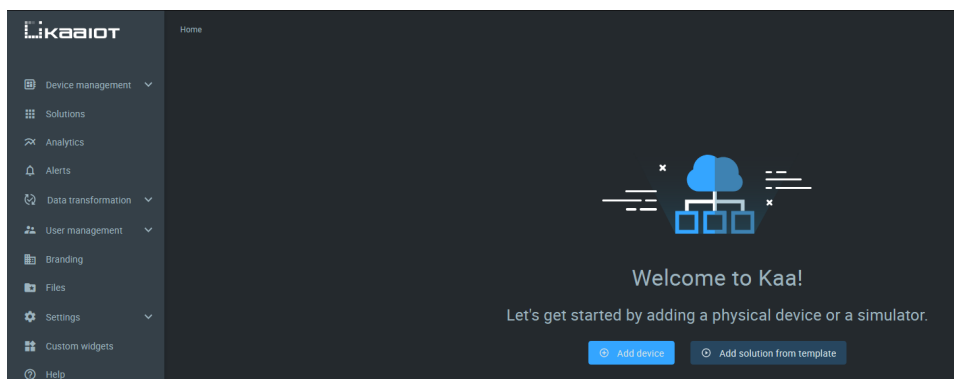


Figura 7.1: Dashboard Kaa IoT.

¹Hace referencia al cuadro de mandos al que tenemos acceso para interactuar con nuestro dispositivo y todas las posibles configuraciones

7.1. Conectar dispositivo

En este apartado se trata de explicar el proceso de conexión de un dispositivo con nuestra aplicación, desde crear un endpoint hasta ver la información del dispositivo en nuestra interfaz de usuario. Esto engloba varios términos y conceptos que se van a definir a continuación. [22]

7.1.1. Términos y conceptos

7.1.1.1. Endpoints

Los endpoints representan “el elemento de las cosas” del IoT. Un endpoint es cualquier dispositivo terminal que se quiera gestionar, en nuestro caso desde Kaa IoT. Un endpoint puede ser un dispositivo físico o una emulación de software del mismo. Todos los datos que llegan a la aplicación están asociados a endpoints. [20]

Para ser precisos, un endpoint puede ser una unidad menor que un dispositivo, lo que significa que un dispositivo físico puede incluir múltiples endpoints. Por ejemplo, quieres gestionar un termostato, para que el aire acondicionado se encienda y apague automáticamente a cierta temperatura.

Se puede gestionar el termostato de una de las siguientes maneras:

- Toda la unidad del termostato actúa como un endpoint único que intercambia datos con el servidor.
- Los componentes del termostato, como los sensores de temperatura y humedad, interruptor de encendido/apagado, actúan como endpoints individuales.

7.1.1.2. ID de Endpoint

El ID de endpoints se utiliza para identificar de forma única un endpoint dentro de una instancia. Un ID de endpoints suele ser un UUID generado automáticamente por el framework en el momento de crear un nuevo endpoint. No obstante, también se permiten los ID de endpoints definidos por el usuario. El ID de los endpoints no puede modificarse una vez creado.

Todos los datos de los endpoints, como los atributos de los metadatos, los puntos de datos de series temporales recopilados, los comandos, etc., están asociados a un ID de endpoint específico. Siempre que recupere o gestione datos relacionados con endpoints en Kaa, principalmente a través de la API REST, se verá los ID de endpoints.

7.1.1.3. Token del Endpoint

Los tokens de endpoints se utilizan para la identificación de endpoints cuando se intercambian datos relacionados con los endpoints, utilizando los protocolos

compatibles basados en MQTT y HTTP. Los tokens de endpoint son únicos dentro de una aplicación IoT y se asignan exactamente a un endpoint.

Cuando llega un mensaje de un cliente, el token del endpoint se resuelve en el correspondiente ID del endpoint. Un ejemplo sobre el protocolo MQTT, el token del endpoint va dentro de la llamada MQTT, por ejemplo:

```
1 kp1/<APPLICATION_VERSION>/epmx/<ENDPOINT_TOKEN>/get
```

Normalmente, los tokens son cadenas generadas automáticamente por el framework, pero también se puede crear un token como el usuario quiera, por ejemplo, por el número de serie del dispositivo, dirección MAC, etc.

7.1.1.4. Metadatos del endpoint

Los metadatos de los endpoints son un conjunto de atributos clave-valor asociados a un endpoint. Se representan en el framework como un documento JSON de formato arbitrario.

Los metadatos de endpoints suelen incluir alguna información relacionada con los endpoints, como la ubicación, la descripción, el número de serie, la versión de hardware, etc. Los metadatos se almacenan en el servicio de registro de endpoints y pueden leerse o actualizarse de dos maneras:

- A través de la capa de comunicación.
- A través de la API REST.

También se pueden gestionar los metadatos mediante la interfaz de usuario del framework.

7.1.1.5. Aplicaciones y versiones

Las aplicaciones en Kaa IoT sirven como contenedores para endpoints de diferentes tipos. Se puede tener una aplicación que contenga todos los endpoints que representan a un determinado dispositivo, y una aplicación para otro dispositivo, independiente de la otra aplicación. Las aplicaciones IoT también albergan toda la configuración del sistema necesaria para que el framework conozca las capacidades de sus dispositivos conectados y cómo trabajar con ellos.

Puede pasar que ya hemos configurado nuestro dispositivo, pero queremos implementar una nueva característica. Al implementarla se actualiza el firmware del dispositivo y se empieza a desplegar pero, ¿cómo diferenciamos entre los dispositivos que ya tienen el nuevo firmware y las que no? Aquí aparecen las versiones de una aplicación.

Cada aplicación puede tener varias versiones al mismo tiempo. Cada versión representa un conjunto de capacidades soportadas por los endpoints. En cualquier momento, cada endpoint está asociado a una versión de su aplicación. El conocimiento de la versión actual de la aplicación de un endpoint ayuda al framework a entender qué funcionalidad soporta el endpoint, cómo se formatean los

datos, etc. Se puede utilizar las versiones para hacer evolucionar los dispositivos añadiendo o retirando funcionalidades mientras mantiene sus versiones antiguas en funcionamiento. Para diferenciar llamadas entre versiones se puede indicar como hemos visto en 7.1.1.3.

7.1.2. Pasos a seguir

7.1.2.1. Crear una aplicación y una versión

Como hemos visto en 7.1.1, para registrar un endpoint en nuestro framework necesitamos una aplicación y una versión de esta. Esto podemos gestionarlo desde la interfaz de usuario, concretamente en la sección “Applications”, una vez en la sección usaremos el botón de “Add application”. Introduciremos el nombre de la aplicación (campo obligatorio) y tendremos la posibilidad de introducir una descripción. En nuestro caso la aplicación se llamará “TFG-DASIoT”.

Hay que tener en cuenta que tanto las aplicaciones como las versiones tienen:

- Nombres autoasignados e inmutables que suelen ser como **7bfdd6b9-ff44-4098-a4dc-58c0f3c9f693-v1**. Se utilizarán para las llamadas a la API, la integración con el cliente, etc.
- Nombres de visualización arbitrarios que se pueden cambiar en cualquier momento. Estos nombres se utilizan en la interfaz de usuario de la plataforma para una mejor experiencia de usuario. Por ejemplo, en nuestro caso el nombre de la aplicación y la descripción que hayamos puesto.



Figura 7.2: Nombre y descripción de la aplicación

En la imagen también se puede observar como hay un identificador justo debajo del nombre que le hemos asignado a la aplicación, esto es para referenciar de manera unívoca a esta. Y ahora que tenemos la aplicación, creamos una versión de esta.

La primera versión que hemos creado se llama “test”. Y en la información de la aplicación podemos ver los servicios que tenemos activos.

- **Data collection.** *epts*, se refiere al servicio de series temporales de endpoints. Recibe muestras de datos de endpoints y los transforma en series temporales. *dcx*, es un servicio de recogida de datos, permite a los endpoints enviar muestras de datos de telemetría a la aplicación.

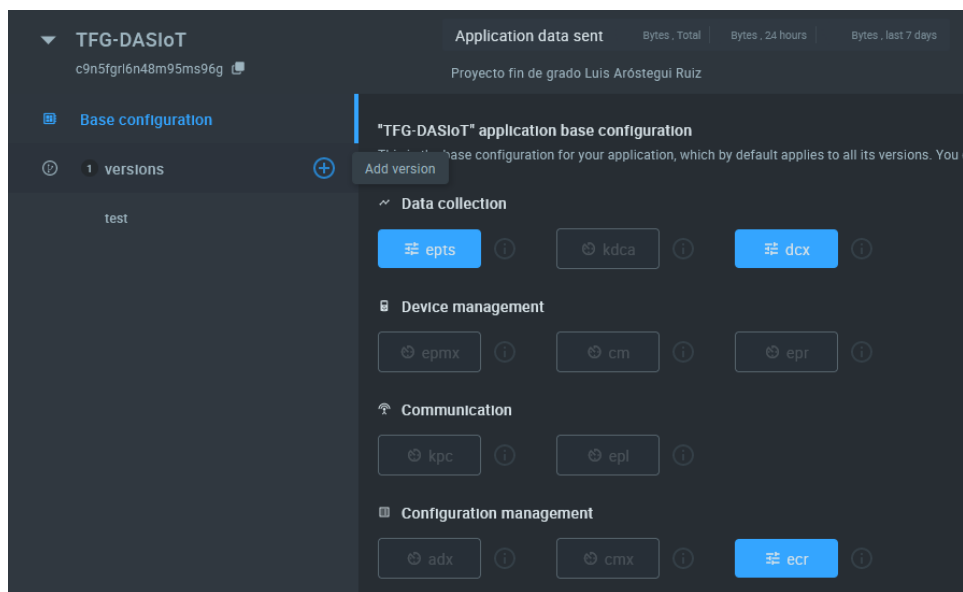


Figura 7.3: Versiones de la aplicación

- **Configuration management.** *ecr*, configuración del repositorio del endpoint, almacena los datos de configuración de los endpoints y proporciona una API REST para la gestión.

7.1.2.2. Crear un endpoint

En la sección de “Devices” de nuestro dashboard, podremos añadir un nuevo dispositivo. Aquí indicaremos, la aplicación a la que va a estar asociada el dispositivo, un nombre para nuestro endpoint y opcionalmente podremos añadir metadatos.

En nuestro caso, para referirnos al endpoint lo hacemos mediante el *token endpoint* 7.1.1.3, que como vemos en 7.4, se llama token1. Esto lo usaremos en nuestra llamada mqtt para hacer referencia a este dispositivo.

Para ver todos los datos del dispositivo se nos muestra como vemos en 7.5. Donde podemos ver la aplicación a la que esta asociada, la fecha de creación y de su última actualización.

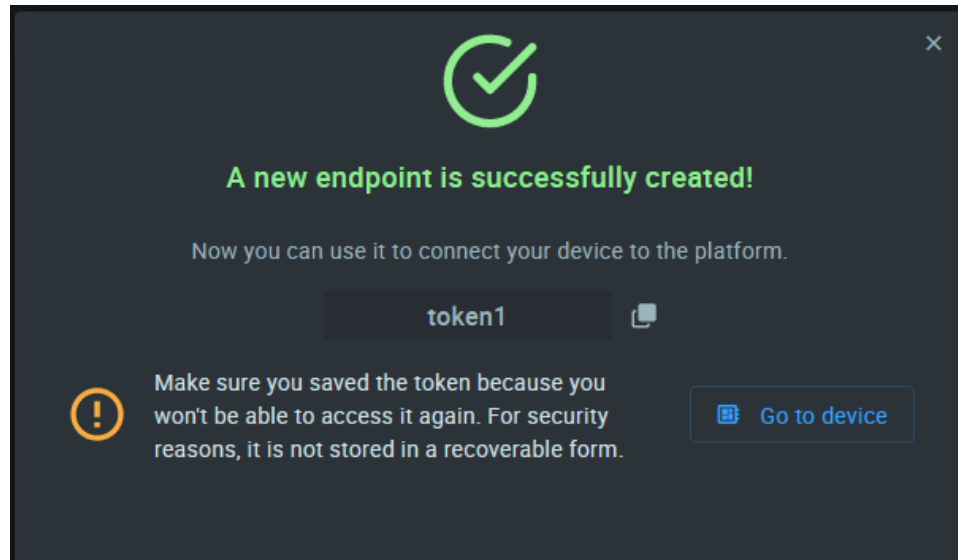


Figura 7.4: Endpoint creado

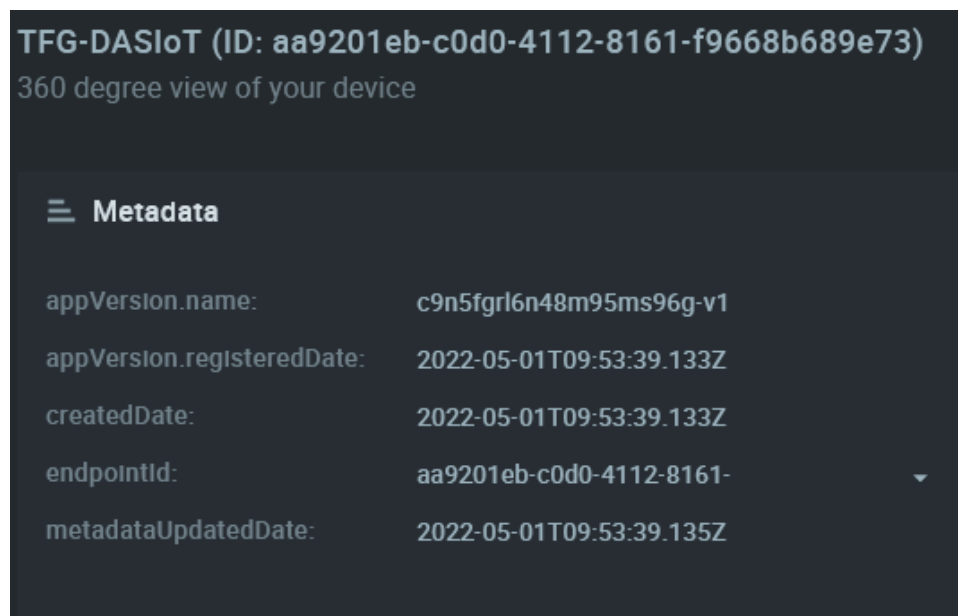


Figura 7.5: Datos del dispositivo creado

7.1.2.3. Conectar un cliente

Una vez ya hemos creado nuestro primer endpoint, podemos conectar un cliente y obtener y enviar algunos metadatos. Como se analizó 5.3, Kaa IoT es un framework que soporta varios protocolos donde nos encontramos con HTTP y MQTT 2.1.2.2.

Para hacer uso de los protocolos y completar la integración del cliente necesitaremos el nombre de la versión y el token del endpoint. En esta sección se va a mostrar como hacer uso de ambos protocolos, concretamente de las órdenes para poder ejecutar la conexión con nuestro dispositivo, en la sección de *Pruebas* 8 se mostrarán los datos que obtenemos tras su ejecución.

7.1.2.3.1. Conexión mediante HTTP

Para obtener todos los atributos de los metadatos con **HTTP** vamos a hacer uso de cURL ² para enviar una solicitud de actualización de datos del dispositivo.

```
1 curl - -location - -request POST 'https://connect.cloud.kaaiot.
   com:443/kp1/
2 <app-version-name>/epmx/<endpoint-token>/update/keys' \
3 - -data-raw '{
4     "model": "BFG 9000",
5     "mac": "00-14-22-01-23-45"
6 }'
```

Ejecutando esta instrucción añadiremos nuevos metadatos a nuestro dispositivo, concretamente la dirección física y el modelo de este.

7.1.2.3.2. Conexión mediante MQTT

Para hacer la conexión mediante **MQTT** se ha optado por la opción de usar Python en su versión 3.10, obtenemos un código como el siguiente.

```
1 import itertools
2 import json
3 import queue
4 import random
5 import string
6 import sys
7 import time
8
9 import paho.mqtt.client as mqtt
10 from decouple import config
11
12 KPC_HOST = config('KPC_HOST', cast=str)
```

²Es un proyecto de software consistente en una biblioteca y un intérprete de comandos orientado a la transferencia de archivos. Soporta los protocolos FTP, FTPS, HTTP, HTTPS, TFTP, SCP, SFTP, Telnet, DICT, FILE y LDAP, entre otros.

```

13 KPC_PORT = config('KPC_PORT', cast=int)
14
15 APPLICATION_VERSION = config('APPLICATION_VERSION', cast=str)
16 ENDPOINT_TOKEN = config('ENDPOINT_TOKEN', cast=str)
17
18
19 class MetadataClient:
20
21     def __init__(self, client):
22         self.client = client
23         self.metadata_by_request_id = {}
24         self.global_request_id = itertools.count()
25         get_metadata_subscribe_topic = f'kp1/{
APPLICATION_VERSION}/epmx/{ENDPOINT_TOKEN}/get/#'
26         self.client.message_callback_add(
get_metadata_subscribe_topic, self.handle_metadata)
27
28     def handle_metadata(self, client, userdata, message):
29         request_id = int(message.topic.split('/')[2])
30         if message.topic.split('/')[1] == 'status' and
request_id in self.metadata_by_request_id:
31             print(f'<--- Received metadata response on topic {
message.topic}')
32             metadata_queue = self.metadata_by_request_id[
request_id]
33             metadata_queue.put_nowait(message.payload)
34         else:
35             print(
36                 f'<--- Received bad metadata response on topic
{message.topic}: \n{str(message.payload.decode("utf-8"))}')
37
38     def get_metadata(self):
39         request_id = next(self.global_request_id)
40         get_metadata_publish_topic = f'kp1/{APPLICATION_VERSION
}/epmx/{ENDPOINT_TOKEN}/get/{request_id}'
41
42         metadata_queue = queue.Queue()
43         self.metadata_by_request_id[request_id] =
metadata_queue
44
45         print(f'--> Requesting metadata by topic {
get_metadata_publish_topic}')
46         self.client.publish(topic=get_metadata_publish_topic,
payload=json.dumps({}))
47         try:
48             metadata = metadata_queue.get(True, 5)
49             del self.metadata_by_request_id[request_id]
50             return str(metadata.decode("utf-8"))
51         except queue.Empty:
52             print('Timed out waiting for metadata response from
server')
53             sys.exit()
54
55     def patch_metadata_unconfirmed(self, metadata):

```

```

56     partial_metadata_udpate_publish_topic = f'kp1/{
APPLICATION_VERSION}/epmx/{ENDPOINT_TOKEN}/update/keys'
57
58     print(f'---> Reporting metadata on topic {
partial_metadata_udpate_publish_topic}\nwith payload {
metadata}')
59     self.client.publish(topic=
partial_metadata_udpate_publish_topic, payload=metadata)
60
61
62 def main():
63     # Inicializar conexion con el servidor
64     print(
65         f'Connecting to Kaa server at {KPC_HOST}:{KPC_PORT}
using application version {APPLICATION_VERSION} and
endpoint token {ENDPOINT_TOKEN}')
66
67     client_id = ''.join(random.choice(string.ascii_uppercase +
string.digits) for _ in range(6))
68     client = mqtt.Client(client_id=client_id)
69     client.connect(KPC_HOST, KPC_PORT, 60)
70     client.loop_start()
71
72     metadata_client = MetadataClient(client)
73
74     # Obtener los atributos de los metadatos del endpoint
actual
75     retrieved_metadata = metadata_client.get_metadata()
76     print(f'Retrieved metadata from server: {retrieved_metadata
}')
77
78     # Actualizar parcialmente los metadatos del endpoint
79     metadata_to_report = json.dumps({"model": "BFG 9001", "mac"
: "00-14-22-02-23-45"})
80     metadata_client.patch_metadata_unconfirmed(
metadata_to_report)
81
82     time.sleep(5)
83     client.disconnect()
84
85
86 if __name__ == '__main__':
87     main()

```

La ejecución de este código produce el mismo efecto que el visto con HTTP. Se usa *decouple* para evitar mostrar los datos de configuración de la aplicación, como el endpoint o la versión de la aplicación (líneas 12-16).

7.2. Recogida de datos de un dispositivo

En esta sección se trata de seguir completando nuestra aplicación, para ello vamos a ver como recoger datos de un dispositivo, visualizar estos datos y como

transformarlos para darles un uso productivo. Antes de empezar con la implementación se definen, como anteriormente, términos y conceptos claves relacionados con el framework para entender el desarrollo. [23]

7.2.1. Términos y conceptos

7.2.1.1. Muestra de datos

Una muestra de datos hay que pensar en ella como un bloque de datos en formato **JSON**, este será enviado por un cliente a la aplicación. Un dispositivo recogerá datos del entorno para el que se haya configurado, estos datos los formateará y los enviará al framework en formato JSON para que sean tratados. Por ejemplo, si tenemos un dispositivo destinado a medir el tiempo meteorológico podremos obtener datos como los siguientes:

```
1 {  
2   "temperature": 25,  
3   "humidity": 46,  
4   "pressure": 800  
5 }
```

7.2.1.2. Series temporales

Las series temporales son una secuencia de puntos de datos con nombre. Cada punto de datos contiene una marca de tiempo y uno o más valores con nombre. Un conjunto de nombres de valores y sus tipos (numérico, string, booleano) define una serie temporal.

Es posible definir diferentes series temporales para varias cosas. Por ejemplo, una serie temporal puede tener sólo un valor numérico o por otro lado, otra serie temporal puede tener varios valores numéricos.

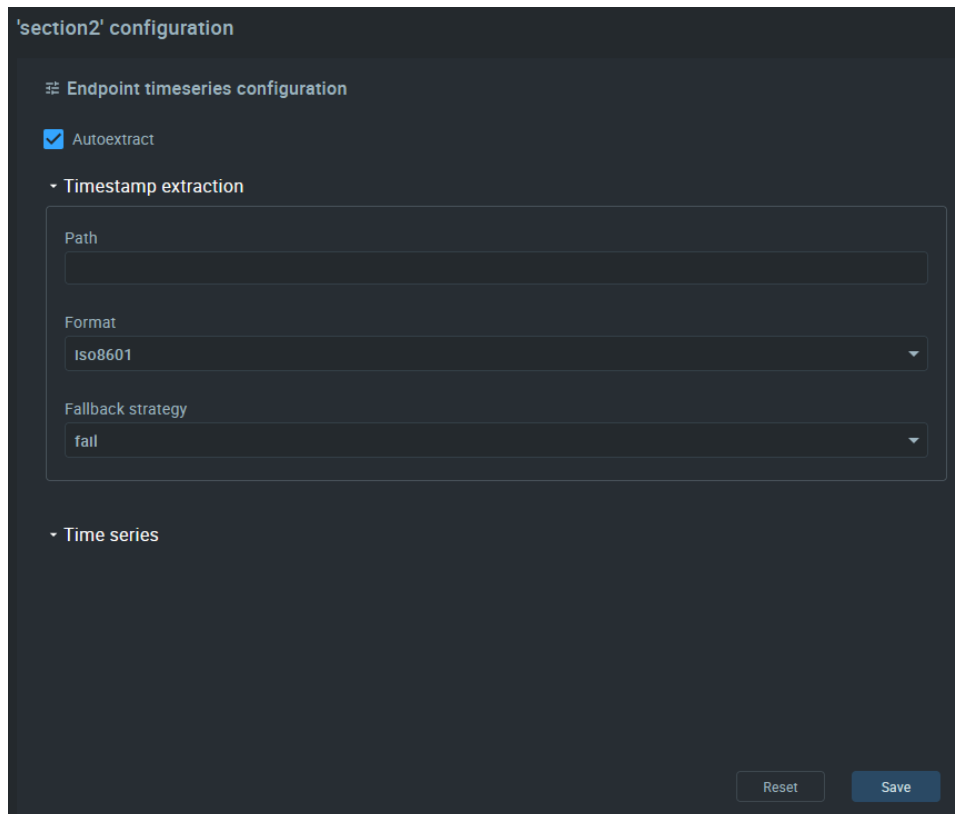
Se puede configurar la aplicación para que transforme las muestras de datos recibidas del endpoint en series temporales para mostrarlas en gráficos, indicadores, mapas, etc. El microservicio responsable de extraer los puntos de datos de las muestras de datos, almacenarlos y recuperarlos, es el servicio Endpoint Time Series (EPTS).

Además, el EPTS tiene una función de auto-extracción que almacena cada campo numérico de muestra de datos de nivel superior en una serie temporal separada. Todas las series temporales auto-extraídas tienen un nombre que sigue el patrón auto <nombre del campo> y un valor numérico con el valor del nombre. Así, si un endpoint envía datos con una muestra con dos campos y la función de auto-extracción está activada, el EPTS crea dos series temporales.

7.2.2. Pasos a seguir

7.2.2.1. Activar la autoextracción de series temporales

Vamos a usar la opción de auto-extracción del EPTS. Para activarla tenemos que ir a la sección de “Gestión de dispositivos” -> “Aplicaciones” -> “EPTS” (en nuestra aplicación) -> Activar la casilla de **Auto extracción**.



The image shows a configuration window titled "'section2' configuration". It contains a section for "Endpoint timeseries configuration" with a checkbox for "Autoextract" which is checked. Below this is a "Timestamp extraction" section with three fields: "Path" (empty), "Format" (set to "Iso8601"), and "Fallback strategy" (set to "fail"). There is also a "Time series" section below. At the bottom right, there are "Reset" and "Save" buttons.

Figura 7.6: Habilitar opción de auto extracción de datos

En nuestro caso hemos creado una versión nueva de nuestra aplicación llamada “section2” para probar esta nueva característica de nuestra aplicación.

Con esta función activada, se crearán automáticamente series temporales para cada campo numérico que se encuentre en la raíz de las muestras de datos que el endpoint envíe. A continuación, podremos ver estas series temporales en la interfaz de usuario del framework, sin necesidad de realizar ninguna configuración adicional.

Ahora en la vista de nuestro dispositivo, concretamente en la sección de

“telemetría del dispositivo”, podremos analizar los datos que recoge nuestro dispositivo.

7.2.2.1.1. Envío de datos mediante HTTP

Es muy similar a como vimos anteriormente en 7.1.2.3.1.

```
1 curl - -location - -request POST 'https://connect.cloud.kaaiot.
    com:443/kp1/
2 <app-version-name>/dcx/<endpoint-token>/json' \
3 - -data-raw '{
4     "intensidad": 50,
5     "color": 48
6 }'
```

7.2.2.1.2. Envío de datos mediante MQTT

Al igual que en la sección 7.1.2.3.2 tenemos un código para hacer el envío de datos. En este caso se excluye la carga de configuración y paquetes importados.

```
1 class DataCollectionClient:
2
3     def __init__(self, client):
4         self.client = client
5         self.data_collection_topic = f'kp1/{APPLICATION_VERSION}
        /dcx/{ENDPOINT_TOKEN}/json'
6
7     def connect_to_server(self):
8         print(f'Connecting to Kaa server at {KPC_HOST}:{
        KPC_PORT} using application version {APPLICATION_VERSION}
        and endpoint token {ENDPOINT_TOKEN}')
9         self.client.connect(KPC_HOST, KPC_PORT, 60)
10        print('Successfully connected')
11
12    def disconnect_from_server(self):
13        print(f'Disconnecting from Kaa server at {KPC_HOST}:{
        KPC_PORT}...')
14        self.client.loop_stop()
15        self.client.disconnect()
16        print('Successfully disconnected')
17
18    def compose_data_sample(self):
19        return json.dumps({
20            'timestamp': int(round(time.time() * 1000)),
21            'intensidad': random.randint(15, 25),
22            'color': random.randint(35, 60),
23        })
24
25
26 def on_message(client, userdata, message):
27     print(f'<-- Received message on topic "{message.topic}":\n{
        str(message.payload.decode("utf-8"))}')
```



```

28
29
30 def main():
31     client = mqtt.Client(client_id=''.join(random.choice(string
        .ascii_uppercase + string.digits) for _ in range(6)))
32
33     data_collection_client = DataCollectionClient(client)
34     data_collection_client.connect_to_server()
35     client.on_message = on_message
36
37     client.loop_start()
38
39     listener = SignalListener()
40     while listener.keepRunning:
41
42         payload = data_collection_client.compose_data_sample()
43
44         result = data_collection_client.client.publish(topic=
            data_collection_client.data_collection_topic, payload=
            payload)
45         if result.rc != 0:
46             print('Server connection lost, attempting to
                reconnect')
47             data_collection_client.connect_to_server()
48         else:
49             print(f'--> Sent message on topic "{
                data_collection_client.data_collection_topic}":\n{payload}'
                )
50
51             time.sleep(3)
52
53             data_collection_client.disconnect_from_server()
54
55
56 class SignalListener:
57     keepRunning = True
58
59     def __init__(self):
60         signal.signal(signal.SIGINT, self.stop)
61         signal.signal(signal.SIGTERM, self.stop)
62
63     def stop(self, signum, frame):
64         print('Shutting down...')
65         self.keepRunning = False
66
67
68 if __name__ == '__main__':
69     main()

```

Con este código se entra en un bucle donde se van haciendo llamadas al dispositivo y recogiendo información. La información que nos llega viene en formato JSON que se especifica en la función *compose_data_sample*.

7.3. Envío de comandos al dispositivo

En esta sección se trata de ejecutar comandos en nuestro dispositivo.

7.3.1. Términos y conceptos

7.3.1.1. Comando

Un comando es un mensaje de corta duración enviado a un endpoint. Con los comandos se pueden encender y apagar las luces o solicitar un informe inmediato del estado de un endpoint.

Cualquier comando puede estar en estado pendiente o ejecutado. El estado pendiente significa que el comando ha sido invocado pero aún no se conoce el resultado de su ejecución. El estado ejecutado se asigna al comando que ha obtenido una respuesta del endpoint, lo que significa que un endpoint recibió el comando, lo ejecutó y envió el resultado de la ejecución a la aplicación.

7.3.1.2. Tipo de comando

Representa el comando que se desea ejecutar en un endpoint, por ejemplo, reiniciar o encender la luz. Un endpoint puede manejar tantos tipos de comandos como se definan en su firmware.

7.3.2. Pasos a seguir

7.3.2.1. Invocar un comando

7.3.2.1.1. Ejecución con HTTP

Cuando se invoca un comando en un dispositivo que se conecta a la aplicación a través de un protocolo **síncrono**, por ejemplo, HTTP, no hay manera de que la plataforma envíe dicho comando al dispositivo. En su lugar, el framework persiste el comando y espera hasta que el dispositivo lo solicite para su ejecución. Esto significa que para los dispositivos con protocolos **síncronos** es nuestra responsabilidad sondear periódicamente la aplicación para nuevos comandos.

Para invocar un comando en la sección de “Dispositivos” hay un cuadro llamado *Ejecución de comandos*. Aquí indicamos el nombre del tipo de comando y una retención máxima. La retención máxima para la entrega define el tiempo en el que el comando está disponible para su ejecución. Una vez rellenado estos campos podemos clicar en “run”. Con esto hemos conseguido que si durante la próxima hora se llama a este comando, se ejecutará en el dispositivo. Para hacer la llamada se puede hacer de la siguiente manera:

```

1 curl --location --request POST 'https://connect.cloud.kaaiot.
   com:443/kp1/<app-version-name>/cex/<endpoint-token>/command
   /<command-name>' \
2 --data-raw '{}'
```

Ejecutándolo, obtendremos como respuesta un ID para identificar de forma única el comando. Con la anterior orden hemos conseguido dejar en *Pendiente*. Para ejecutar el comando, mandaremos la siguiente orden:

```

1 curl --location --request POST 'https://connect.cloud.kaaiot.
   com:443/kp1/<app-version-name>/cex/<endpoint-token>/result
   /<command-name>' \
2 --data-raw ' [{
3     "id": <command-ID>,
4     "statusCode": 200,
5     "reasonPhrase": "OK",
6     "payload": "Success"
7 } ]'
```

Cuando recibe la aplicación el resultado de la ejecución del comando con el ID, marca este como *Ejecutado*.

7.3.2.1.2. Ejecución con MQTT

Con este protocolo el proceso se simplifica. Simplemente tenemos que dejar ejecutando el siguiente código y desde nuestro framework indicamos el tipo de comando y acción sobre el dispositivo.

```

1 class DataCollectionClient:
2
3     def __init__(self, client):
4         self.client = client
5         self.data_collection_topic = f'kp1/{APPLICATION_VERSION}
6         }/dcx/{ENDPOINT_TOKEN}/json/32'
7
8         command_reboot_topic = f'kp1/{APPLICATION_VERSION}/cex
9         /{ENDPOINT_TOKEN}/command/reboot/status'
10        self.client.message_callback_add(command_reboot_topic,
11        self.handle_reboot_command)
12        self.command_reboot_result_topik = f'kp1/{
13        APPLICATION_VERSION}/cex/{ENDPOINT_TOKEN}/result/reboot'
14
15        command_zero_topic = f'kp1/{APPLICATION_VERSION}/cex/{
16        ENDPOINT_TOKEN}/command/zero/status'
17        self.client.message_callback_add(command_zero_topic,
18        self.handle_zero_command)
19        self.command_zero_result_topik = f'kp1/{
20        APPLICATION_VERSION}/cex/{ENDPOINT_TOKEN}/result/zero'
21
22    def connect_to_server(self):
23        print(f'Connecting to Kaa server at {KPC_HOST}:{
24        KPC_PORT} using application version {APPLICATION_VERSION}
25        and endpoint token {ENDPOINT_TOKEN}')
```

```

17         self.client.connect(KPC_HOST, KPC_PORT, 60)
18         print('Successfully connected')
19
20     def disconnect_from_server(self):
21         print(f'Disconnecting from Kaa server at {KPC_HOST}:{KPC_PORT}...')
22         self.client.loop_stop()
23         self.client.disconnect()
24         print('Successfully disconnected')
25
26     def handle_reboot_command(self, client, userdata, message):
27         print(f'<--- Received "reboot" command on topic {message.topic} \nRebooting...')
28         command_result = self.compose_command_result_payload(message)
29         print(f'command result {command_result}')
30         client.publish(topic=self.command_reboot_result_topik, payload=command_result)
31         # With below approach we don't receive the command confirmation on the server side.
32         # self.client.disconnect()
33         # time.sleep(5) # Simulate the reboot
34         # self.connect_to_server()
35
36     def handle_zero_command(self, client, userdata, message):
37         print(f'<--- Received "zero" command on topic {message.topic} \nSending zero values...')
38         command_result = self.compose_command_result_payload(message)
39         client.publish(topic=self.data_collection_topic, payload=self.compose_data_sample(0, 0, 0))
40         client.publish(topic=self.command_zero_result_topik, payload=command_result)
41
42     def compose_command_result_payload(self, message):
43         command_payload = json.loads(str(message.payload.decode("utf-8")))
44         print(f'command payload: {command_payload}')
45         command_result_list = []
46         for command in command_payload:
47             commandResult = {"id": command['id'], "statusCode": 200, "reasonPhrase": "OK", "payload": "Success"}
48             command_result_list.append(commandResult)
49         return json.dumps(command_result_list)
50
51     )
52
53     def compose_data_sample(self, fuelLevel, minTemp, maxTemp):
54         return json.dumps({
55             'timestamp': int(round(time.time() * 1000)),
56             'fuelLevel': fuelLevel,
57             'temperature': random.randint(minTemp, maxTemp),
58         })
59

```

```

60
61 def on_message(client, userdata, message):
62     print(f'Message received: topic {message.topic}\nbody {str(
        message.payload.decode("utf-8"))}')
63
64
65 def main():
66     # Initiate server connection
67     client = mqtt.Client(client_id=''.join(random.choice(string
        .ascii_uppercase + string.digits) for _ in range(6)))
68
69     data_collection_client = DataCollectionClient(client)
70     data_collection_client.connect_to_server()
71
72     client.on_message = on_message
73
74     # Start the loop
75     client.loop_start()
76
77     fuelLevel, minTemp, maxTemp = 100, 95, 100
78
79     # Send data samples in loop
80     listener = SignalListener()
81     while listener.keepRunning:
82
83         payload = data_collection_client.compose_data_sample(
            fuelLevel, minTemp, maxTemp)
84
85         result = data_collection_client.client.publish(topic=
            data_collection_client.data_collection_topic, payload=
            payload)
86         if result.rc != 0:
87             print('Server connection lost, attempting to
            reconnect')
88             data_collection_client.connect_to_server()
89         else:
90             print(f'--> Sent message on topic "{
            data_collection_client.data_collection_topic}":\n{payload}'
            )
91
92             time.sleep(3)
93
94             fuelLevel = fuelLevel - 0.3
95             if fuelLevel < 1:
96                 fuelLevel = 100
97
98             data_collection_client.disconnect_from_server()
99
100
101 class SignalListener:
102     keepRunning = True
103
104     def __init__(self):
105         signal.signal(signal.SIGINT, self.stop)

```

```
106         signal.signal(signal.SIGTERM, self.stop)
107
108     def stop(self, signum, frame):
109         print('Shutting down...')
110         self.keepRunning = False
111
112
113 if __name__ == '__main__':
114     main()
```

Luis Aróstegui Ruiz

CAPÍTULO 8

Pruebas

CAPÍTULO 9

Conclusiones y trabajos futuros

Bibliografía

- [1] Preeti Agarwal and Mansaf Alam. Investigating iot middleware platforms for smart application development. In *Smart Cities—Opportunities and Challenges*, pages 231–244. Springer, 2020.
- [2] Haider Al-Shammari. <https://www.researchgate.net/publication/338166577/figure/fig1/as:840165425676289@1577322455370/soa-based-architecture-for-iot-middleware-1.jpg>, consultado el 11 de Abril de 2022.
- [3] Apigee. <https://www.zettajs.org/>, consultado el 17 de Abril de 2022.
- [4] Rusu Liviu Dumitru. Iot platforms: Analysis for building projects. *Informatica Economica*, 21(2), 2017.
- [5] Applied Informatics Software Engineering GmbH. <https://macchina.io/>, consultado el 17 de Abril de 2022.
- [6] D. Hanes, G. Salgueiro, P. Grossetete, R. Barton, and J. Henry. *IoT Fundamentals: Networking Technologies, Protocols, and Use Cases for the Internet of Things*. Fundamentals. Pearson Education, 2017.
- [7] Wan Haslina Hassan et al. Current research on internet of things (iot) security: A survey. *Computer networks*, 148:283–294, 2019.
- [8] Altair Engineering Inc. <https://www.altair.com/smartworks/>, consultado el 17 de Abril de 2022.
- [9] IoTDunia. <https://iotdunia.com/what-is-iot-framework-list-top-10-open-source-iot-frameworks/#what-is-iot-framework>, consultado el 15 de Abril de 2022.
- [10] Farhana Javed, Muhamamd Khalil Afzal, Muhammad Sharif, and Byung-Seo Kim. Internet of things (iot) operating systems support, networking

- technologies, applications, and challenges: A comparative review. *IEEE Communications Surveys Tutorials*, 20(3):2062–2100, 2018.
- [11] labplus. https://mpython.readthedocs.io/en/master/_images/three-layer-iot-architecture.png, consultado el 11 de Abril de 2022.
 - [12] P. Lea. *IoT and Edge Computing for Architects: Implementing Edge and IoT Systems from Sensors to Clouds with Communication Systems, Analytics, and Security, 2nd Edition*. Expert insight. Packt Publishing, 2020.
 - [13] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications. *IEEE internet of things journal*, 4(5):1125–1142, 2017.
 - [14] Bhumi Nakhuva and Tushar Champaneria. Study of various internet of things platforms. *International Journal of Computer Science & Engineering Survey*, 6(6):61–74, 2015.
 - [15] Anne H. Ngu, Mario Gutierrez, Vangelis Metsis, Surya Nepal, and Quan Z. Sheng. lot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, 4(1):1–20, 2017.
 - [16] RF Page. <https://www.rfpage.com/what-are-the-major-components-of-internet-of-things/>, consultado el 15 de Abril de 2022.
 - [17] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. Middleware for internet of things: A survey. *IEEE Internet of Things Journal*, 3(1):70–95, 2016.
 - [18] D. Serpanos and M. Wolf. *Internet-of-Things (IoT) Systems: Architectures, Algorithms, Methodologies*. Springer International Publishing, 2018.
 - [19] Sajjad Hussain Shah and Ilyas Yaqoob. A survey: Internet of things (iot) technologies, applications and challenges. In *2016 IEEE Smart Energy Grid Engineering (SEGE)*, pages 381–385, 2016.
 - [20] KaaloT Technologies. <https://docs.kaaiot.io/kaa/docs/current/kaa-concepts>, consultado el 1 de Mayo de 2022.
 - [21] KaaloT Technologies. <https://www.kaaiot.com/>, consultado el 17 de Abril de 2022.
 - [22] KaaloT Technologies. <https://docs.kaaiot.io/kaa/docs/current/tutorials/getting-started/connecting-your-first-device/>, consultado el 8 de Mayo de 2022.
 - [23] KaaloT Technologies. <https://docs.kaaiot.io/kaa/docs/current/tutorials/getting-started/collecting-data-from-a-device/>, consultado el 9 de Mayo de 2022.

- [24] Inc Temboo. <https://temboo.com/iot>, consultado el 18 de Abril de 2022.
- [25] Inc. The MathWorks. <https://thingspeak.com/>, consultado el 17 de Abril de 2022.
- [26] BK Tripathy and J. Anuradha. *Internet of Things (IoT): Technologies, Applications, Challenges and Solutions*. CRC Press, 2017.