Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Trabajo Fin de Máster

Diseño de un protocolo de descubrimiento de servicios e intercambio de mensajes basado en coreografía para sistemas empotrados distribuidos en el "Edge" de la arquitectura IoT

Autor: Senén Melquiades Palanca Barrio

Tutor: Dra. Sara Blanc Clavero

Curso 2020 - 2021

Resumen

Este documento presenta una solución bajo el paradigma de Internet de las Cosas (IoT) que permite el descubrimiento de microservicios distribuidos en la capa Edge de la arquitectura IoT. La capa Edge de la arquitectura IoT está compuesta por máquinas Raspberry Pi. Cada máquina ofrece diferentes microservicios organizados según un sistema colaborativo de coreografía. La solución propuesta implementa un middleware que incorpora modelos de transmisión de mensajes para que sean coherentes y compatibles con los protocolos actuales de mensajería de IoT. Estos modelos de transmisión, junto con un protocolo de descubrimiento basado en el estándar de OASIS WS-Discovery, permiten conectar e interrelacionar nuevos microservicios ubicados en distintas máquinas de forma rápida y efectiva, así como ayudar con el equilibrio de la carga de CPU en las distintas máquinas. El descubrimiento se implementa como un microservicio que actúa de intermediario (bróker) y admite expresiones regulares (Regexp) en el contenido del mensaje para interpretar tanto los patrones de publicación ofrecidos por los microservicios proveedores de datos como las necesidades de servicios consumidores de datos.

Palabras clave: Internet de las cosas, sistemas de coreografía, Rasberry Pi, Edge Computing, WS-Discovery, lenguaje natural, Regexp, aplicación, riego

Abstract

This paper presents a solution to support service discovery for edge choreography based distributed embedded systems. The Internet of Things (IoT) edge architectural layer is composed of Raspberry Pi machines. Each machine hosts different services organized based on the choreography collaborative paradigm. The solution adds to the choreography middleware three messages passing models to be coherent and compatible with current IoT messaging protocols. It is aimed to support blind hot plugging of new machines and help with service load balance. The discovery mechanism is implemented as a broker service and supports regular expressions (Regexp) in message scope to discern both publishing patterns offered by data providers and client services necessities. Results compare Control Process Unit (CPU) usage in a request–response and datacentric configuration and analyze both Regexp interpreter latency times compared with a traditional message structure as well as its impact on CPU and memory consumption.

Keywords : Internet of Things, Choreography Computing Systems, Rasberry Pi, Edge Computing, Internet WS-Discovery, Natural Language, Regexp, App, Irrigation.

Tabla de contenidos

1. Introducción

El ámbito informático de Internet de las Cosas está en constante crecimiento. La evolución de las comunicaciones y el aumento de la potencia de los dispositivos empotrados ha hecho posible la creación de arquitecturas orientadas a recoger y ofrecer datos de forma eficiente y cada vez con menos retardo de carga y procesado. Actualmente, la mejora de los dispositivos empotrados, o de bajas prestaciones, hace posible que asuman funcionalidades de cómputo que, hasta ahora, sólo eran propias de los dispositivos desplegados en la capa "Cloud" o en nube. Mientras que hasta hace poco las "cosas" sólo servían como terminales, cada vez sirven más como nodos con capacidad de computación; es lo que se denomina capa Edge o Fog de la arquitectura IoT (Internet of Things (IOT): An overview and its applications, 2017).

La idea inicial de Internet de las Cosas distinguía entre los dispositivos que recopilan los datos con sus sensores y el servidor central que procesa los datos y toma decisiones. Actualmente, existe cierta tendencia a trasladar parte de las decisiones que se toman en el servidor central a las capas Edge y Fog como capas intermedias. Así se reduce el tráfico del sistema. Además, se mejora la eficiencia general ya que el trabajo se distribuye entre los nodos delegando sobre ellos parte de la capacidad de decisión, se reducen el tiempo de transmisión desde la fuente hasta la toma de decisiones y es posible diseñar sistemas especializados independientes. Este concepto se denomina Edge Computing (Design, Resource Management, and Evaluation of Fog Computing Systems: A Survey, 2021), (What is Edge computing?) y será un objeto de análisis en la solución presentada.

El presente trabajo propone una solución novedosa que aprovecha las ventajas de muchas de las últimas tecnologías diseñadas para arquitecturas IoT. En concreto, se pretende aportar una solución al descubrimiento de nuevos servicios en una arquitectura SOA (Service Oriented Architecture) basada en coreografía de servicios (Integration of Distributed Services and Hybrid Models Based on Process Choreography to Predict and Detect Type 2 Diabetes. Sensors, 2017). En concreto, los servicios se encuentran distribuidos entre varias máquinas de tipo Raspberry Pi. Para el desarrollo de esta solución, se parte como base de un motor de coreografía desarrollado por el equipo SABIEN (SABIEN, 2021) del Instituto ITACA-UPV. Este motor recientemente se ha adaptado para realizar implementaciones de sistemas IoT orientadas a microservicios. Se trata de un software potente y ligero que permite la ejecución en paralelo de microservicios y su intercomunicación. Está implementado con las últimas versiones de .NET Core 3, por lo que es compatible con Windows IoT para Raspberry Pi, el hardware utilizado en este proyecto.

El motor de coreografía permite que servicios implementados en C# se ejecuten de manera asíncrona e independiente entre sí. Cada máquina ejecuta un motor de coreografía y el motor tiene la capacidad de conectarse con otros motores y ofrecer un sistema único a ojos de los microservicios, aunque estén ejecutándose en máquinas distintas.

Por tanto, los microservicios se alojan en distintas máquinas y se comunican intercambiando mensajes. Cada mensaje enviado por un microservicio es distribuido y recibido por todos los microservicios del sistema de coreografía, independientemente de la ubicación física de la máquina y microservicio. En la recepción se admite el filtrado y selección de mensajes basado en máscaras sobre el identificador del mensaje.

El motor fue creado siguiendo las recomendaciones de la Fundación para Agentes Físicos Inteligentes (FIPA) (FIPA@ Towards a Standard for Software Agents, 1998). La sección de cabecera de mensaje se extrae del Protocolo simple de acceso a objetos (SOAP) (W3C). En caso de necesitar serializar los mensajes, se admiten tanto XML como JSON, pero también posibles extensiones.

En este TFM se ha propuesto la implementación de un middleware sobre este motor. El middleware está compuesto por un conjunto de microservicios que tienen como objetivo desplegar un sistema escalable y centrado en ofrecer datos de una forma dinámica a otros

microservicios. Cada máquina Raspberry Pi del sistema de coreografía actúa como Gateway de una WSN (Wireless Sensor Network). Cada Gateway recibe datos de diferentes sensores siendo capaz de almacenarlos y pre-procesarlos para ofrecerlos a otras Gateways: microservicios proveedores. A su vez, las Gateways también implementan microservicios de toma de decisiones o gestión que requieren datos tanto de sí misma y de otras máquinas: microservicios consumidores.

Los proveedores y los consumidores necesitan intercambiar datos mediante algún mecanismo o patrón de intercambio en sus mensajes. Según las necesidades del sistema, se han implementado tres patrones de intercambio: REQ-RES, DATACENTRIC e INTELIGENTE.

Estos modelos funcionarán siempre y cuando los microservicios se reconozcan entre sí. Por eso ha sido necesario implementar un microservicio de descubrimiento o bróker. El bróker permitirá que nuevos microservicios que se conecten al sistema "en caliente", se registren y aporten los datos necesarios acerca del tipo de microservicio y sus posibles patrones de intercambio. El bróker se encarga de realizar la clasificación de los nuevos microservicios y también su activación en el caso de los proveedores de tipo DATACENTRIC e INTELIGENTE. Además, también se encarga de la asociación de microservicios: proveedor-consumidor, priorizando los patrones DATACENTRIC e INTELIGENTE para reducir, así, el tráfico del sistema, como se explicará en las posteriores secciones.

En el TFM se ha realizado un análisis de escalabilidad, a fin de observar la carga de CPU y memoria que supone el bróker en función de distintas configuraciones de microservicios.

Finalmente, se ha desarrollado una app que permite recibir y observar los datos generados en el sistema.

1.1. Motivación

Intentaremos en este capítulo justificar las motivaciones -tanto personales como generales que nos han hecho interesarnos en este proyecto. Desde un punto de vista personal, la motivación para elección del tema viene de lejos. Cuando desarrollé el TFG con la Dra. Sara Blanc, hicimos una primera aproximación a lo que sería este TFM.

La elección del tema viene motivada por una doble faceta. Por una parte, pretendíamos huir de la hiperespecialización de dispositivos que hace que el usuario final se encuentre en un cúmulo de tecnologías que a veces se solapan e interactúan de manera negativa o, al menos, no eficiente. Buscábamos desarrollar una solución inteligente y eficaz que permitiera interconectar dispositivos no complejos entre sí y, a la par, ofrecer un sistema homogéneo independiente de la estructura de servicios.

Por otra parte, no hay que olvidar la utilidad y el usuario final del proyecto: se trata de un programa para escuelas, para facilitar el desarrollo de los huertos educativos a la par que conseguir que la gestión sea lo más efectiva y ecológica posible. La administración de los recursos será tanto más eficiente cuanto los datos proporcionados a los usuarios sean precisos, consiguiendo de esta manera el ahorro en los gastos y recursos necesarios para el mantenimiento de un huerto. Pero creemos que el desarrollo del proyecto pueda ir más allá de esta aplicación. Las soluciones propuestas para la comunicación, gestión y almacenamiento de datos podrán ser útiles en otros contextos relacionados con IoT y este proyecto, o al menos el sistema de coreografía con dispositivos de bajo coste podría servir como base para resolver nuevos problemas en este ámbito.

A nivel personal, el proyecto me ha supuesto, desde el principio, un reto ya que implicaba no solo poner en práctica aquellos diversos conocimientos adquiridos a lo largo de los estudios,

sino ir un paso más allá y aprender otras tecnologías y poner en práctica, en el mundo real, los problemas con los que nos podemos enfrentar en el desarrollo de un proyecto de grandes dimensiones.

Por otra parte, tecnología y sostenibilidad deben ir de la mano. La implementación de sistemas informáticos en las diversas áreas del saber, como la Agricultura o la Ecología, será una herramienta no solo válida sino necesaria y nuestro trabajo abre la puerta a la colaboración con esos ámbitos. El proyecto no debe quedarse en papel mojado porque los avances hemos desarrollado, la posibilidad de utilizar una aplicación móvil para gestionar las características de un determinado huerto -en este caso escolar-. a bajo coste, se puede extrapolar a muchas más actividades.

1.2. Objetivos

Los objetivos del proyecto son los siguientes:

1. Adaptar el motor de coreografía ofrecido por el grupo SABIEN del instituto ITACA-UPV a un sistema compuesto por varias máquinas de tipo Raspberry Pi.
2. Desarrollo de un middleware sobre motor de coreografía que permita el intercambio directo de mensajes entre microservicios. Entre los objetivos que nos proponemos, es imperativo realizar un análisis en profundidad de las formas de descubrimiento y de intercambio de información o mensajería IoT. Queremos que nuestra solución sea coherente y compatible con los sistemas IoT y estándares más utilizados.
3. Desarrollo de un microservicio de descubrimiento sobre el middleware que actúe de bróker en el descubrimiento facilitando los datos necesarios a los microservicios consumidores para activar el intercambio de mensajes por eventos puntuales o continuo. El bróker se basará en el estándar de OASIS WS- Discovery. Además, será necesario probar el sistema implementado y realizar varios análisis de rendimiento y uso de memoria para determinar la escalabilidad del sistema.
4. Dotar al bróker de un intérprete de lenguaje natural con Regexp que permita identificar las necesidades de un consumidor o las posibilidades de un proveedor en cuanto al patrón de intercambio, datos requeridos/ofrecidos, frecuencia de envío, etc. El uso de lenguaje natural nos va a permitir generar una semántica simplificada en los mensajes para reducir su tamaño en el cuerpo del mensaje.
5. Desarrollo de un software Android capaz de visualizar por medio de gráficas e histogramas los datos recabados por sensores. La información utilizada por esta aplicación estará almacenada en un servidor de bases de datos externo al sistema IoT, en concreto se trata de Firebase. Por tanto, otro objetivo del TFM es desarrollar un microservicio capaz de enviar los datos necesarios a Firebase para que la aplicación Android pueda funcionar.

1.3. Colaboraciones

Todo lo que contiene este proyecto se ha publicado en un artículo de investigación en el que participan dos doctorados de la UPV. Nestor Xavier Arreaga Alvarado aportó un sistema de pruebas para comparar a nivel físico los retardos de los mensajes. Jose Luís Bayo Montón aportó el motor del Coreógrafo sobre el que se ejecuta el middleware desarrollado en este TFM.

La parte del Proyecto relacionada con el objetivo del desarrollo de software Android ha sido financiada por el proyecto eSGarden: School Gardens for Future Citizens. Number: 2018-ES01-

KA201-050599. Erasmus + KA201 de 2018, coordinado por la Universitat Politècnica de València[1].

Nuestro trabajo no es el resultado de una idea o una propuesta individual. El TFM extiende la solución desarrollada en el artículo (A Service Discovery Solution for Edge Choreography-Based Distributed Embedded Systems, 2021), que debe sus bases a la iniciativa de mi tutora y mentora, la doctora Sara Blanc, coautora e impulsora del artículo citado.

[1] https://ec.europa.eu/programmes/erasmus-plus/projects/eplus-project-details/#project/2018-1-ES01-KA201-050599

2. Estado del arte

Las últimas innovaciones en tecnología y comunicación permiten la adaptación del paradigma de Internet de las Cosas (IoT) a gran cantidad de áreas de aplicación. Está comprobado que una arquitectura en capas beneficia la adaptación; podemos encontrar varios ejemplos que proponen capas interconectadas que abarcan desde los sensores hasta el usuario final (Survey of platforms for massive IoT, 2018). La complejidad de estas capas depende de los objetivos de aplicación y la magnitud de sistema a observar, la localización y situación de los sensores y actuadores y la infraestructura disponible. Son muchos los beneficios en multitud de contextos si se aprovechan adecuadamente las ventajas de una arquitectura IoT. Entre las más destacables se incluye la existencia de una capa de computación Edge o Fog capaz de llevar a cabo almacenamiento de datos, gestión y microcontrol especializada de sistemas físicos, toma de decisiones, integración de microservicios externos y la intra e interoperabilidad entre sistemas con distintos intereses.
Sin embargo, construir una capa de computación de tipo Edge con dispositivos de bajo coste es un desafío. La capacidad computacional de estos dispositivos o máquinas suele ser muy limitada en comparación con el alto rendimiento de los sistemas en la nube. Por tanto, en los sistemas Edge IoT, las máquinas necesitan intercambiar información para tomar decisiones mientras el almacenamiento se mantiene distribuido.

Actualmente existen dos tendencias para describir cómo debe fluir la información. Se trata de la orquestación y de la coreografía de microservicios (Web Services Orchestration and Choreography, 2003). Por un lado, la orquestación encaja en un modelo centralizado donde un elemento de cómputo central gestiona el estado del proceso general del sistema y el flujo de datos (Complementing IoT Services Through Software Defined Networking and Edge Computing: A Comprehensive Survey, 2020). Por otro lado, la coreografía se basa en una combinación de microservicios distribuidos que cooperan para aportar funcionalidad al sistema y administrar los procesos y el flujo de datos. Esto hace que la coreografía sea apropiada para aplicarla a microservicios de tipo Edge y situados distribuidamente en máquinas con capacidades computacionales similares y normalmente no muy elevadas.

En relación a la coreografía de microservicios, en los últimos años, se han presentado varias plataformas para modelar y componer microservicios bajo el paradigma de la coreografía: [CHOReOS (Improving Health Monitoring With Adaptive Data Movement in Fog Computing, 2020), (A Universal Complex Event Processing Mechanism Based on Edge Computing for Internet of Things Real-Time Monitoring, 2019)] y su evolución a CHOReVOLUTION (CHOReVOLUTION: Service choreography in practice, 2020); ActnConnect (Hollosy); ChorSystem (ChorSystem: A Message-Based System for the Life Cycle Management of Choreographies, 2016); y otros trabajos [(Cruz-Filipe, et al., 2019), (Evaluating IoT service composition mechanisms for the scalability of IoT systems, 2020), (Savanovic, et al., 2020), (: A tool chain for choreographic design, 2020), (Coto-Santiesteban, et al., 2020)] además de algunas extensiones aplicadas a modelado de microservicios: [(Hahn, et al., 2017), (Modeling Data Transformations in Data-Aware Service Choreographies, 2018), (Dynamic IoT Choreographies, 2019), (Choreographing Services for Smart Cities: Smart Traffic Demonstration, 2017)].
Sin embargo, el uso de la coreografía en la capa Edge de IoT aún no está muy extendida, aunque tiene gran potencial en sistemas profesionales comerciales. Probablemente porque no está bien definida la utilidad, eficiencia y escalabilidad para su uso en sistemas distribuidos con máquinas de bajas prestaciones.

Destacamos TraDE (Hahn, y otros, 2017). Por lo general, los modelos de coreografía combinan el control y el flujo de datos dentro del flujo de trabajo del sistema. Para desacoplar el flujo de datos y el modelado de flujo de control, TraDE es un middleware que admite flujos de datos entre microservicios mediante el uso de objetos de datos y un almacenamiento intermedio. Los trabajos anteriores de los autores de TraDe incluyen la plataforma ChorSystem [(ChorSystem: A Message-Based System for the Life Cycle Management of Choreographies,

2016)], un entorno de coreografía. Sin embargo, en ese entorno los datos se asumen permanentes, cuando a nivel de Edge el almacenamiento de datos tiende a ser efímero. Los datos generados por los sensores son útiles durante un corto período de tiempo y son necesarios casi inmediatamente tras su obtención. Por tanto, TraDe es un gran ejemplo de una aplicación que gestiona datos distribuidos y que también está basada en la coreografía.

Otro ejemplo es el trabajo presentado en el artículo (Dynamic IoT Choreographies, 2019) que utiliza un modelo de coreografía para asociar a los consumidores de datos con diferentes proveedores. El enfoque se centra en ofrecer una configuración dinámica. Por ejemplo, es posible cambiar la asociación entre luces e interruptores en una habitación sin modificar el equipo instalado y la electrificación. Sin embargo, el sistema es estático y no incorpora descubrimiento ni conexión de nuevos microservicios en tiempo de ejecución.

Finalmente, el trabajo en el artículo (Choreographing Services for Smart Cities: Smart Traffic Demonstration, 2017) presenta CHOReVOLUTION para gestionar información en tiempo real en el enrutamiento del tráfico. El enfoque es prometedor en los sistemas cooperativos de transporte inteligente donde los vehículos, la infraestructura y numerosos microservicios en la nube están conectados y cooperan para brindar soluciones de transporte eficientes. Sin embargo, no se aborda el descubrimiento de microservicios en tiempo real.

Por tanto, podemos decir que la coreografía de microservicios aplicada a redes de máquinas de bajas prestaciones en la capa Edge del IoT es una línea poco explorada pero que, a priori, representa una tendencia e importantes ventajas en el despliegue de sistemas de apoyo a la nube. Por todo esto, y dada la cantidad y complejidad de conceptos relativos al área de la informática que aplica, creo que ha sido una gran oportunidad para explorar, aprender y "crear" trabajando con estándares, plataformas, modelos y lenguajes que me van a servir en mi desempeño profesional.

3. Crítica al estado del arte

El motor de coreografía utilizado en este TFM se explica en el artículo (Semantic Process Choreography for Distributed Sensor Management., 2010). Este motor permite establecer rutas que comuniquen los microservicios entre sí. Además, utiliza de manera eficiente los recursos del sistema, de tal manera que resulta adecuado para dispositivos IoT (Wearable Sensors Integrated with Internet of Things for Advancing eHealth Care, 2018). El motor de coreografía se despliega de forma individual en cada máquina Raspberry Pi, aunque todas las máquinas con el mismo motor se interconectan como si de una única máquina se tratara. Cada máquina aloja un subconjunto de microservicios. La conexión de los microservicios es transparente a su ubicación física y a la capa de transporte.

Sin embargo, las rutas se establecen de forma estática. El motor de coreografía utilizado no incluye ningún mecanismo de descubrimiento que permita establecer las rutas dinámicamente. Pero este mecanismo es necesario para permitir la conexión "en caliente", o en tiempo de ejecución, de nuevas máquinas con nuevos microservicios. Cuando se conecta una nueva máquina al sistema, los microservicios alojados en esa máquina no conocen a priori la identificación de ningún otro microservicio de la red con el que se puedan conectar para intercambiar datos. Este proyecto propone la implementación de ese microservicio de descubrimiento.

Además, el motor de coreografía no se diseñó para IoT. En este TFM se quiere adaptar el motor para que sea coherente con las actuales soluciones IoT para posibles extensiones que lo hagan compatible.

Por tanto, en el TFM nos enfrentamos a tres problemas de implementación. Por un lado, necesitamos modelos de intercambio de datos en los mensajes similares a la mensajería típica IoT. Aunque el motor de coreografía ya implementa dos formas básicas de mensajería,

tendremos que adaptarlas y extenderlas. Segundo, necesitamos un mecanismo de descubrimiento que no existe aún. Tercero, queremos una prueba de concepto que incluya un visor para observar los datos enviados por los sensores y así disponer de un prototipo completo.

En cuanto al **primer problema**, nuestra solución admite tres modelos: REQ-RES, DATACENTRIC e INTELIGENTE. Veamos el porqué:

- El tipo REQ-RES, define la interacción más simple entre microservicios, en la que un microservicio consumidor solicita datos a un proveedor por medio de una petición directa esperando una respuesta también directa.
- El tipo DATACENTRIC, que se utiliza cuando uno o varios microservicios necesita los mismos datos de un proveedor de forma periódica y desea subscribirse a él.
- Por último, el tipo INTELIGENTE, que permite la subscripción a una regla ofrecida por un proveedor. Por ejemplo, cuando el envío de los datos se quiere limitar al envío de aquellos que sobrepasen un umbral, o al máximo de un intervalo de tiempo, o a la desviación de una tendencia, entre otros. Por tanto, el proveedor inteligente incorpora un procesado que reduce el número de envíos al consumidor. Tras la subscripción, el envío de los datos que cumplen la regla se realiza de forma directa entre el proveedor de la regla de datos y el consumidor.

Los dos primeros patrones de comunicación son especialmente relevantes para el intercambio de mensajes en IoT. Por ejemplo, el "Constrained Application Protocol" (CoAP) (Shelby, y otros, 2014) es un protocolo de transferencia web basado en el Protocolo de transferencia de hipertexto (HTTP) que admite el patrón de comunicación de solicitud-respuesta (REQRES). El "Message Queuing Telemetry Transport Protocol" (MQTT) (Internet of Things: Survey and open issues of MQTT Protocol, 2017) y el MQTT-SN para redes de sensores (OASIS) utilizan también solicitud-respuesta y subscripción a tópicos. El "Advanced Message Queuing Protocol" (AMQP) (A comparison of AMQP and MQTT protocols for Internet of Things, 2019) admite mensajes tanto de solicitud-respuesta como de publicación-suscripción basada en tópicos. El "Extensible Messaging and Presence Protocol" (XMPP) admite solicitud-respuesta, mensajería asincrónica y publicación-suscripción[2]. Por último, el "Data Distribution Service"[3] admite la comunicación centrada en datos. El protocolo "DataCentric Publish-Subscribe" (DCPS) de DDS entrega información a los receptores no en función de su destino, sino en función del tipo de datos que contiene el mensaje. Por lo tanto, el patrón centrado en datos o DATACENTRIC es un modelo de comunicación de uno a muchos. La subscripción a tópicos puede implementarse como una forma de distribución de mensajes identificados por su contenido.

En resumen, teniendo en cuenta que los patrones de intercambio más comunes son REQ-RES y DATACENTRIC, y si nuestra solución pretende ser compatible con otros sistemas a través de un Proxy, tendremos que incorporarlos en nuestra solución.
Además, también atendiendo al uso que se espera de este middleware en microservicios asociados a sensores hemos implementado un nuevo tipo que denominamos INTELIGENTE, basado en la detección de una regla. Cuando la regla se cumple, se dispara el evento asíncrono que detona el envío de los datos. El gestor de reglas no se ha implementado en el TFM, pero sí el patrón de comunicación.

En segundo lugar, necesitamos un mecanismo de descubrimiento de microservicios. En este caso, existe el estándar de OASIS WS-Discovery[4]. Los flujos de mensajes utilizados en el estándar son fácilmente adaptables a cualquier dispositivo de un sistema basado en SOA. Pero existen dos inconvenientes. En primer lugar, nuestros mensajes ya tienen una cabecera forzosa y necesaria para establecer las rutas acorde con el motor de coreografía. Por tanto, el formato de los mensajes puede ser compatible pero no idéntico.

[2] https://xmpp.org/uses/internet-of-things.html
[3] https://www.omg.org/omg-dds-portal/
[4] http://docs.oasis-open.org/ws-dd/discovery/1.1/wsdd-discovery-1.1-spec.html

La propuesta de este TFM es utilizar el estándar para copiar el flujo del protocolo y una estructura similar en las tramas. De forma que, en caso de querer, en el futuro, conectar el Edge de coreografía con un sistema nativo WS-Discovery, el Proxy pueda "acomodar" los mensajes de entrada y salida de forma automática y sencilla.
Además, en el estándar, en caso de utilizar bróker, este se encarga tanto del descubrimiento como de la gestión de rutas. Esta es otra diferencia con este TFM ya que, el bróker sólo se encargaría del descubrimiento. Las rutas se gestionan con el motor de coreografía de forma distribuida entre todas las máquinas y no centralizada en este microservicio.

Por otro lado, otro problema del estándar es el tamaño de los mensajes que se generan. La anotación semántica incluye muchos caracteres de control y líneas. Necesitamos simplificarlos. Con máquinas con escasos recursos de memoria y cómputo, una implementación WS nativa nos produciría una saturación rápida de los buffers de gestión de mensajes.
Por eso, necesitamos una solución que nos permita acortar los mensajes pero manteniendo la estructura del formato WS-Discovery.
Hay que tener en cuenta que aún no se ha explorado el uso de expresiones regulares en sistemas empotrados integrado en el descubrimiento de microservicios. Nuestra solución utiliza un análisis de expresiones regulares (Regexp) para discernir entre los patrones de publicación ofrecidos por un proveedor de datos y las peticiones con las necesidades del cliente. Utilizando lenguaje natural, hemos podido crear una anotación semántica propia en los mensajes, reduciendo líneas y caracteres de control.

Por último, para que la solución desarrollada tenga una componente de visualización de los datos recogidos en tiempo real, desarrollaremos una aplicación Android (app). Esta app debe conectar con un microservicio de base de datos de respaldo en nube sobre la que lee y escribe valores. En concreto, este TFM ha desarrollado una app específica para agricultura de precisión. El motivo es ligar el TFM con un proyecto Erasmus + que actualmente lidera la UPV en colaboración con cuatro colegios europeos, tres universidades, un centro de investigación para agricultura avanzadas y una empresa tecnológica especializada en agricultura. Este proyecto (A Service Discovery Solution for Edge Choreography-Based Distributed Embedded Systems, 2021) tiene entre sus objetivos, desarrollar una app que permita la gestión educativa de huertos urbanos y escolares. Por eso, los valores leídos de los sensores se representan en gráficas diarias y catalogadas según espacio y sensor. El espacio no se refiere a la localización geográfica, que se podría hacer, sino a la organización del huerto.

Queremos una solución sencilla, accesible al público en general, pero que de varias funcionalidades como son: organizar espacios, añadir y eliminar nuevos cultivos, añadir o eliminar sensores asociados a cultivos, trabajar con sensores multiparamétricos, representarlos en el eje del tiempo, poder visualizar los datos de diferentes formas y con zoom, generar alertas sobre umbrales, y otros detalles.

Se han presentado algunos trabajos previos de desarrollo de apps en la UPV, especialmente en agricultura [(A Smart Decision System for Digital Farming, 2019), (Boukharouba, 2020)]. En nuestro caso, esta app no es el objetivo primero ni único del proyecto, sino una tarea más.

En cuanto a las plataformas de desarrollo de la app, actualmente Adroid Studio es una de las más utilizadas. Entre los lenguajes que soporta encontramos Dart con la extensión de Flutter que permiten interconectar de forma rápida y segura dispositivos móviles con servicios de Google Cloud. Flutter es la nueva apuesta de Google para las aplicaciones móviles, web y de escritorio. Las ventajas que supone utilizar esta arquitectura han permitido que el desarrollo de la aplicación sea muy rápido y eficiente. El hecho de usar Flutter confiere a la aplicación la capacidad de ejecutarse en la red, por lo que ya no sólo se trata de una aplicación de tipo 'cross-platform' para móviles, sino que ahora también se puede ejecutar en un servidor y visualizarse en una página web sin realizar ningún cambio sobre el proyecto desarrollado para móvil.

También de Google, se ha elegido Firebase como servicio de base de datos. Esta base de datos ofrece, como ventaja, una visualización de los datos por consola que nos ha ayudado mucho en los tests de depuración. Sobre Firebase, podemos añadir que se ha seleccionado por

su versión gratuita y porque es sencilla para manejar por un público poco experto a nivel de usuario. Como Firebase es la BD de nuestros colegios, la elegimos por eso, para que ellos pudieran manejarla y conectarla con dispositivos tipo Arduino sin muchas dificultades.

4. Análisis del problema

Desplegar un motor de coreografía sobre dispositivos de bajo coste y limitación computacional implica ciertas consideraciones a la hora de utilizar de forma adecuada los recursos que estas máquinas ofrecen. Las arquitecturas de tipo Edge son muy novedosas y aportan beneficios relevantes, pero tienen limitaciones en potencia de cómputo y memoria. Si se superan estas dificultades, se obtiene un sistema versátil y flexible que servirá para adaptarse bien a los requisitos de nuestros futuros sistemas IoT.

Mediante el análisis de los aspectos más importantes de IoT se propone una solución en forma de modelo de microservicios.

Ejemplos de algunos microservicios son la gestión de la red de sensores, la gestión de electroválvulas de riego, el envío de datos a la base de datos de soporte o la gestión de alarmas entre otros. Además, existen microservicios de apoyo como son la gestión de la base de datos local en cada máquina, los conectores de red o los filtros de datos que se aplican para evitar el almacenamiento de datos erróneos.

La problemática relacionada con la baja capacidad de las máquinas no es algo que deba menospreciarse. De hecho, es esencial realizar un modelado correcto de los microservicios y de las interacciones que se producen entre ellos para evitar posibles situaciones no deseadas.

El contenido de este apartado se ha dividido en cuatro secciones:

- En primer lugar, explicamos la arquitectura IoT y su diseño por capas. Especialmente describimos la capa Edge y Fog. Pondremos un ejemplo del uso de la capa Edge en agricultura de precisión o agricultura digital.

- En segundo lugar, explicaremos el contexto del desarrollo realizado en el TFM sobre un motor de coreografía. Los microservicios de la capa Edge se han implementado sobre este motor, pero no directamente sino con un middleware que describimos en esta sección.

- En tercer lugar, analizaremos los distintos patrones de comunicación entre microservicios y en qué se diferencian.

- En cuarto lugar, analizaremos detenidamente el protocolo WS-Discovery en el que se apoya la solución desarrollada en el middleware para la gestión del publicación y descubrimiento de nuevos microservicios. Además, explicaremos el mecanismo utilizado para interpretar patrones de comunicación. Esta funcionalidad combina Regexp y diversos algoritmos de análisis y clasificación de expresiones que permiten interpretar el contenido de los mensajes.

En apartados posteriores se explicarán los microservicios básicos que vamos a necesitar en nuestro sistema adicionales a los consumidores y proveedores de datos.

4.1. Análisis del problema

Entre los objetivos más importantes de una arquitectura IoT se encuentra la capacidad de adaptarse a un sistema real para mejorar su funcionamiento y tomar decisiones a gran escala que de otra forma sería imposible realizar. El uso de esta tecnología se focaliza en que una gran cantidad dispositivos de bajo coste interactúen entre sí y con sistemas complejos. De este modo se crea una red sobre la que se puede tomar decisiones, además de explotar con gran nivel de precisión todas las características o funcionalidades que tiene el sistema. La capa de red es la piedra angular de los sistemas IoT.

Además, el rápido crecimiento que ha sufrido esta tecnología junto con el creciente número de sectores interesados en aprovecharla ha llevado a las empresas a integrarla cada vez más en sus sistemas, hasta el punto de que, hoy en día ya se trata de una pieza fundamental en la industria. Básicamente, los sistemas IoT se encargan de agregar inteligencia a la solución inicial, facilitar su funcionamiento y automatizar las funcionalidades que ofrece.

Debido a que no existe una definición universal de lo que significa IoT, la implementación de las arquitecturas IoT ha dejado durante la última década diferentes propuestas [(Jabraeil Jamali, y otros, 2019), (An Internet of Things (IoT) Architecture for Smart Agriculture, 2018), (Intelligent Agriculture and Its Key Technologies Based on Internet of Things Architecture, 2019)]. La mayoría de estas propuestas se basan en una arquitectura por capas. Desde que Cisco presentó por primera vez la idea de las capas Edge/Fog computing, estas dos capas han mostrado un gran interés por las ventajas que pueden proporcionar en los sistemas IoT y además, no son incompatibles con la arquitectura ISO de referencia (ISO).

En todas las aproximaciones existe una red de sensores en el sistema que permite recoger datos del mundo real y monitorizar los procesos existentes en él, pero varía la visión de la estructura general del sistema, cómo se organizan sus elementos y el nivel de profundidad en la especificación.

4.1.1. Estructura atendiendo a la red de comunicaciones

La primera aproximación parte de un modelo por capas que presenta 3 niveles y se centra en un enfoque técnico acerca de cómo pueden estar organizados los elementos de un sistema IoT a nivel de red de comunicaciones. Estos niveles son la capa de percepción, la de red y la de la aplicación. La capa de percepción está formada por los “things”, que se encargan de recabar los datos del mundo físico mediante distintos tipos de sensores. La capa de red es la capa central en esta arquitectura y su función es transmitir y procesar la información que recibe de la capa de percepción. A su vez, esta capa es la responsable de interconectar distintas redes de dispositivos, servidores, etc. Por último, la capa de aplicación es la encargada de ofrecer a los usuarios la funcionalidad que brinda el sistema IoT. En resumen, esta aproximación ofrece una visión de la estructura de las comunicaciones de un sistema IoT, pero deja por definir la lógica de las capas, sus interacciones en un mayor nivel de detalle, etc. Es por ello, que surge la siguiente aproximación.

4.1.2. Estructura atendiendo a los componentes

Se define con mayor nivel de detalle respecto a los componentes de un sistema IoT. En esta aproximación se entiende que un Sistema IoT no sólo se reduce al ámbito de internet o las redes de comunicación, sino también considera esenciales todos aquellos procesos para limpiar, procesar, almacenar y transmitir información entre diferentes entidades. Por eso se añade a las capas anteriores una capa de transporte, una capa de procesamiento y otra de negocio. La capa de transporte es la encargada de implementar la recepción de información de los 'things'. Habilita diferentes tipos de conexiones tales como redes 3g-4g, Wifi, LoRa™ o ZigBee, entre otras. La capa de procesamiento filtra los datos recibidos y realiza análisis sobre ellos. En esta capa se pueden aplicar técnicas de Big Data. Por último, la capa de negocio ofrece una solución final a los clientes.

4.1.3. ¿Qué es el Edge y cómo se considera en este proyecto?

Es necesario definir el término 'Edge', ya que puede llevar a confusiones. En concreto, este término que proviene de 'Edge Computing', ofrece una visión distribuida en la que las aplicaciones que utilizan los datos están mucho más cerca de estos. Esto significa que la distancia entre los sensores y los centros de procesamiento de la información es mucho menor que si estuvieran en la nube. El procesado de los datos ahora ya no se realiza en grandes centros de cómputo, sino que se delega a pequeñas máquinas en un entorno distribuido. Como se comentaba anteriormente, el término Edge, que fue acuñado por Cisco, ofrece una visión distinta a las estructuras tradicionales. Esta visión, según se aplique, puede mejorar las latencias, facilitar las comunicaciones y añadir mayor seguridad al sistema en general. Pero para que esto sea posible se cuenta con un nuevo elemento: el nodo 'Niebla'. Este nodo tiene capacidades de cómputo y se sitúa más cerca del origen de los datos. Las características más destacables de este tipo de nodos son:

- Se trata de nodos autónomos. Tienen la suficiente capacidad como para tomar decisiones de forma local.
- Son administrables y programables. Es decir, se pueden gestionar desde alguna capa superior.
- Se despliegan en múltiples entornos y no dependen de una estructura de infraestructura concreta para poder funcionar.
- Aportan distintas funcionalidades internas, tales como seguridad y encriptación, visualización de datos, pre-procesado de los datos, o un almacenamiento local, entre otras funciones.

Según los requisitos del sistema sobre el que se desea actuar habrá aproximaciones que convengan más o menos. En nuestro caso, se pretende desplegar un sistema de microservicios heterogéneo que permita aumentar el número de microservicios y que genere un tráfico de información a través de la red de forma controlada. Este sistema de microservicios se ejecutará sobre máquinas consideradas de baja capacidad computacional. En concreto, esta capa estará formada por máquinas Raspberry pi que albergarán un número variable de microservicios y que ofrecerán sus servicios locales a las demás Raspberrys pi del Edge. El sistema de microservicios se dividirá en servicios de apoyo, servicios que consumen datos y servicios que proveen datos. En concreto, estos microservicios se encargan de recoger datos del entorno (red de sensores, datos de otro servidor) y actuar sobre los datos con algún objetivo concreto.

Sobre este conjunto de microservicios se modela el intercambio de información basado en descubrimiento con un bróker.

4.2. Motor de coreografía y el middleware propuesto

La coreografía al nivel del Edge tiene como objetivo distribuir la carga computacional entre las máquinas que componen el sistema. Los dispositivos ubicados en la capa Edge pueden albergar múltiples microservicios de distintos tipos, independientes, pero que colaborarán entre sí. Por ejemplo, como ya hemos comentado anteriormente, en la solución propuesta se ha planteado un microservicio que se comunica con Firebase, un servidor de base de datos en nube, con el objetivo de ofrecer datos a una aplicación móvil desarrollada con Flutter.

Pero para que, en una arquitectura orientada a microservicios, un servicio pueda enviar un mensaje a otro, es necesario que un middleware sea capaz de gestionar ese proceso. En nuestro caso, el middleware utilizado es el motor de coreografía, que es invisible para los servicios, pero es la entidad que gestiona las rutas y permite el intercambio de mensajes mediante el uso de identificadores y máscaras. El motor de coreografía se ejecuta "por debajo" de los microservicios del sistema.

Por ejemplo, la figura 2 muestra una vista general de un ecosistema de microservicios. En el centro de cada máquina está representado un motor de coreografía. Desplegados entre las diferentes máquinas existen microservicios de interconexión entre redes de sensores (WSN) [(Exploiting smart e-Health gateways at the edge of healthcare Internet-of-Things: A fog computing approach, 2018) (Middleware for Smart Gateways Connecting Sensornets to the Internet, 2010)] y el sistema de coreografía: los Gateways. Estos microservicios de tipo Gateway reciben datos que se almacenan en las bases de datos locales ubicadas una en cada máquina. También existen microservicios que se utilizan para añadir nuevas máquinas que tengan el motor de coreografía al sistema distribuido: los microservicios TCP/IP o UDP/IP. Además, se pueden incorporar microservicios de proxy, o microservicios que interoperan con otros microservicios externos (Automated synthesis of mediators for middleware-layer protocol interoperability in the IoT, 2019). Generalmente, en el sistema se distinguen dos tipos de microservicios de forma general, los proveedores y los consumidores de la información, que están representados en la figura 1. Sobre este sistema se representa un microservicio especial que ofrece funcionalidades especiales tales como el descubrimiento o la suscripción. Este microservicio es el bróker.

El bróker permite el descubrimiento de microservicios y gestiona las suscripciones entre ellos. Inicialmente se había considerado la necesidad de utilizar al bróker como intermediario de los mensajes, pero para aliviar su carga, en esta solución el intercambio de mensajes se realiza sin la intervención del bróker.

Además, este microservicio permitirá determinar las reglas de carga de los microservicios. Por ejemplo, si un cliente necesita datos de temperatura en un área donde hay dos o más sensores iguales, el bróker podrá seleccionar el que más conviene atendiendo al tráfico y carga de las máquinas. Esta funcionalidad no se ha implementado en el TFM, pero queda preparado para realizarse en el futuro.

En la Figura 1 vemos que sólo hay un bróker que dará microservicio a todas las máquinas y sus microservicios. En futuras versiones se podrían plantear réplicas del microservicio de bróker para tolerar fallos en el sistema o para generar grupos dentro de la red local y repartir mejor la carga asociada al bróker (clusters).

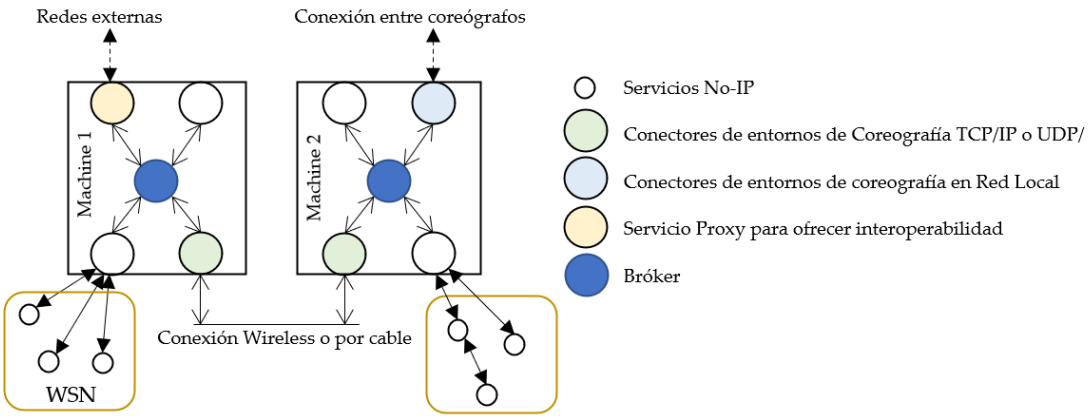

Figura 1. Sistema de coreografía al nivel del Edge.

Pero para que los microservicios se comuniquen e intercambien mensajes necesitan patrones de comunicación adecuados. En la Figura 2, cada máquina aloja un motor de coreografía idéntico, y cada motor está conectado a los motores de las otras máquinas. El motor de la Figura 2 está representado en verde y se desarrolló en el instituto ITACA de la UPV.

El motor de coreografía implementado en este trabajo está basado en el estándar SOAP que define cómo dos microservicios pueden comunicarse mediante el intercambio de información estructurada en XML. En nuestro caso, se ha utilizado una estructura de mensajería basada en la ontología FIPA (Foundation for Intelligent Physical Agents), definida por la organización W3C, que especifica una serie de campos obligatorios en la cabecera de los mensajes entre microservicios. Entre los campos que debe contener la cabecera, se encuentran el identificador del mensaje, el identificador del microservicio origen y el identificador del microservicio destino. El motor utilizará estos campos para redireccionar un mensaje al microservicio correspondiente pudiendo también hacer broadcasts y realizar mensajes de tipo petición-respuesta donde el microservicio proveedor ya tenga preparado parcialmente el mensaje de respuesta.

En este TFM se han desarrollado los elementos representados en naranja en la Figura 2 sobre el motor de coreografía descrito. En concreto, una capa de descubrimiento y tres patrones de comunicación entre microservicios.

En el sistema de coreografía, los microservicios no conocen cuál es la localización de los microservicios con los que comunican; es decir, no saben (ni les importa) la ubicación física del microservicio receptor del mensaje. Por tanto, como el servicio no sabe en qué máquina está el servicio con el que desea comunicarse, cada mensaje se reenvía a todos los coreógrafos del sistema, que entregarán el mensaje a los microservicios internos de cada máquina y que posteriormente y de forma individual comprobarán si deben o no procesarlo. Los mensajes tienen una dirección de destino que es el identificador de enrutamiento. Así, un microservicio recibe el mensaje cuando el identificador del mensaje coincide con uno de su lista de identificadores de recepción. Además, se pueden enviar mensajes a varios microservicios mediante el uso de máscaras sobre los identificadores. Aprovechando esta característica, podemos hacer envíos de uno a varios o de uno a uno.

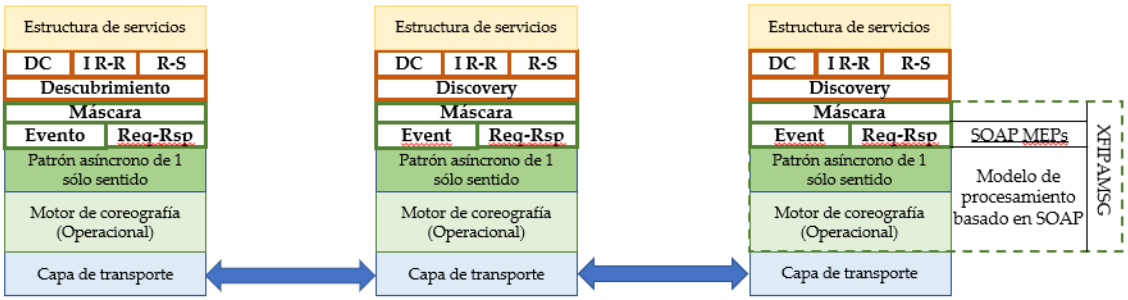

Figura 2. Una estructura en capas apoyada por un motor de coreografía.

El motor de coreografía de la Figura 2 admite tanto el modelo de mensajería basado en solicitud-respuesta (modelo "request-response" o REQ-RES) como una mensajería de eventos asincrónica (modelo de comunicación por eventos) basada en SOAP. En este caso, un microservicio envía un mensaje cuando se produce cierto evento en ese microservicio.

El motor se ha construido siguiendo un estilo de arquitectura en capas que acepta nuevas capas para agregar nuevos patrones de intercambio, permitiendo así una mayor flexibilidad a la hora de adaptarse a futuras tecnologías.
En la Figura 2 se representan los tres patrones utilizados por nuestro bróker: el patrón DATACENTRIC (DC en la Figura 2), o comunicación de datos de uno a varios. Un ejemplo de protocolo que utiliza el patrón Datacentric en IoT es el protocolo Data Distribution Service (DDS) (OASIS). Este patrón nos ayudará a soportar la subscripción de varios microservicios consumidores a los datos enviados por el mismo microservicio proveedor. El patrón de publicación-suscripción para reglas (R-S en la Figura 2) al que hemos llamado INTELIGENTE en este TFM; el consumidor requiere datos de un proveedor, pero este sólo le enviará los datos cuando se produzca un evento determinado. Además, mantenemos el modelo de solicitud-respuesta (IR-R en la Figura 2) que ya está por defecto en la arquitectura de coreografía y llamamos REQ-RES en este TFM. Este último modelo sólo permite la comunicación de uno a uno.

A continuación, se describe el microservicio de descubrimiento o bróker y los patrones de comunicación con más detalle.

4.3. Patrones de intercambio de mensajes

En nuestro sistema existen tres patrones de comunicación de la información. La arquitectura propuesta distingue una entidad que juega un papel muy importante en el sistema, el bróker. Todos los patrones de intercambio de información utilizan de alguna forma este elemento ya que es el único servicio que inicialmente conocen todos los demás servicios del sistema y sobre el que se realiza el proceso de descubrimiento cuando un nuevo microservicio se conecta. Consecuentemente deberá ser el único que conozca en todo momento los microservicios que se encuentran en ejecución y los canales de información que se han desplegado.

Aunque ya se han presentado los distintos modelos de datos que acepta nuestro sistema, ahora veamos con mayor detalle qué características tiene cada uno, qué beneficios aporta y cuáles son sus diferencias.

Los patrones de transmisión de datos se han definido partiendo del análisis de las dificultades que surgen en la comunicación entre entidades. Cada uno de los patrones intenta solventar un problema físico real. Se han identificado las necesidades más importantes de nuestro sistema como la posibilidad de realizar una petición y recibir una respuesta de un

microservicio, suscribirse a un microservicio que envía datos de forma periódica, o suscribirse a un canal de información inteligente, donde fluye información que ya ha sido filtrada y procesada por filtros o reglas de aplicación. ¿Cómo interviene el bróker en cada caso?

Para el primero de los casos, el patrón REQ-RES, se establece una comunicación de punto a punto. Los microservicios que utilicen este patrón podrán solicitar datos a otro microservicio con su mismo patrón. En este caso el bróker sirve de gran utilidad para que el servicio consumidor de datos sepa qué servicios existen con este patrón y a cuáles puede realizar una petición, según las necesidades que tenga.

En segundo lugar, los microservicios pueden recibir datos de forma periódica mediante el patrón DATACENTRIC. En este patrón, el servicio proveedor de datos realiza envíos cada cierto tiempo constante. El bróker tiene un papel más importante que en el patrón anterior ya que sabe en todo momento qué tipo de información envía el servicio proveedor (información que obtiene a través del proceso de descubrimiento) y cada cuanto realiza los envíos para ofrecer esa información a los servicios que necesiten suscribirse al servicio proveedor de datos. Además, es tarea del bróker detectar cuándo se produce una desconexión de todos los microservicios suscritos a un microservicio proveedor de datos para que este último pueda dejar de emitir datos y evitar así una sobrecarga en las comunicaciones del sistema.

Por último, los microservicios que utilizan el patrón INTELIGENTE tienen una serie de parámetros que condicionan que se produzca el envío de los datos. En este modelo de comunicación el microservicio proveedor de datos ofrece los datos sólo y cuando se cumpla una regla. En el futuro, los parámetros de esa regla se configurarán según las necesidades del microservicio que consume los datos. En este patrón el bróker tiene un rol muy importante. El bróker deberá enviar el correspondiente mensaje al microservicio proveedor indicando los parámetros de la regla especificados por el consumidor. Se evita la comunicación directa entre consumidor y proveedor porque no es útil que el consumidor conozca la dirección del emisor de datos ya que sólo la utilizaría para especificarle una regla. Después de que el microservicio consumidor de datos finalice el proceso de descubrimiento, el bróker le facilitará un canal o identificador al que se debe registrar para recibir la información con ese identificador. Hay que recalcar que el microservicio consumidor actuará únicamente como receptor de los datos por el canal que le ha facilitado el bróker y nunca emitirá información.

En resumen, hemos implementado tres patrones muy flexibles que permiten realizar casi cualquier tipo de interacción entre microservicios.

4.4. Protocolo oasis WS Discovery

Para facilitar la futura interoperabilidad con otros sistemas, el mecanismo de descubrimiento está inspirado en el protocolo estándar OASIS de Web Services Dynamic Discovery[5] que define una ontología útil para estructurar el contenido de los mensajes de manera que otros sistemas con el mismo estándar puedan comunicarse con el nuestro.

Este estándar ofrece una especificación de campos a utilizar ubicados en una estructura conocida. Su objetivo es facilitar la comunicación entre sistemas que no tienen la misma infraestructura. Sin embargo, como se menciona en (A Study of LoRa: Long Range & Low Power Networks for the Internet of Things, 2016) o (Automated synthesis of mediators for middleware-layer protocol interoperability in the IoT, 2019), implementar este estándar directamente en dispositivos empotrados no siempre es tarea fácil debido a que se necesita una potencia de cómputo algo elevada y nuestros dispositivos tienen muy poca capacidad.

[5] http://docs.oasis-open.org/ws-dd/discovery/1.1/wsdd-discovery-1.1-spec.html

Este estándar está definido por la organización OASIS (*Organization for the Advancement of Structured Information Standards*). Esta organización es un consorcio internacional que busca la creación de protocolos comunes en sistemas enfocados a ofrecer microservicios por Internet. Su objetivo principal es facilitar la interoperabilidad entre sistemas a través de la adopción de nuevos estándares. En concreto, WS-Discovery se centra en ofrecer un protocolo de descubrimiento para un número elevado de microservicios ubicados en la misma red mediante técnicas de multicast. Los microservicios no conocen la dirección de los otros microservicios por lo que utilizan este protocolo para darse a conocer o para descubrir otros microservicios. En concreto, hemos adoptado la visión del protocolo que se basa en la interpretación de los mensajes por medio de un bróker. Este bróker recibe peticiones de microservicios con una necesidad y es el mismo bróker quien debe atender las necesidades respondiendo con la dirección del microservicio que más se adecúe al caso concreto. Para ello, previamente el microservicio que ofrece el bróker debe de haberse registrado en él.

Probablemente uno de los problemas más importantes que presenta OASIS sobre la ontología de WS Discovery es la estructura de los mensajes. El tamaño de un mensaje aumenta considerablemente cuando un microservicio ofrece varios recursos y también las necesidades de cómputo para procesarlo. Por eso, en este TFM se ha adaptado el protocolo con el objetivo de hacerlo más ligero.

A pesar de que los microservicios están dispersos en distintas máquinas (Raspberry Pi) y solo una de ellas contiene el bróker, en la Figura 3 se presenta la vista lógica del sistema.

Figura 3: Los datos de los sensores se almacenan entre todas las máquinas que forman la capa Edge. Cada máquina almacena los datos de los sensores que se reciben en esa máquina. El proceso de descubrimiento se realiza por un microservicio único para todo el sistema, el bróker, físicamente ubicado en una sola máquina. Este principio mantiene la coherencia y simplifica el sistema implementado en nuestro TFM. En el futuro se podría mejorar la eficiencia generando clústers con un bróker asociado por clúster.

En un sistema donde los datos se almacenan en pequeñas unidades de memoria, el intercambio de mensajes se centra en ofrecer los datos de cada máquina a las demás. Cada microservicio puede ser un proveedor o un consumidor, o ambos.

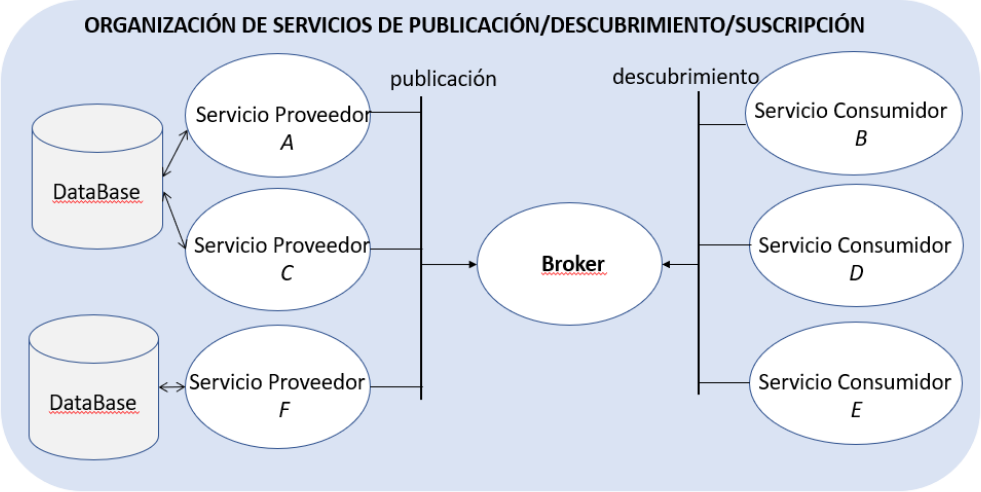

Figura 3. Nivel de abstracción de microservicios lógicos.

La Figura 4 muestra una vista general del algoritmo de descubrimiento basado en WS-Discovery. Los mensajes HELLO y BYE son parte del mecanismo de publicación, mientras que PROBE, PROBE MATCH, RESOLVE y RESOLVE MATCH son parte del protocolo de descubrimiento y subscripción.

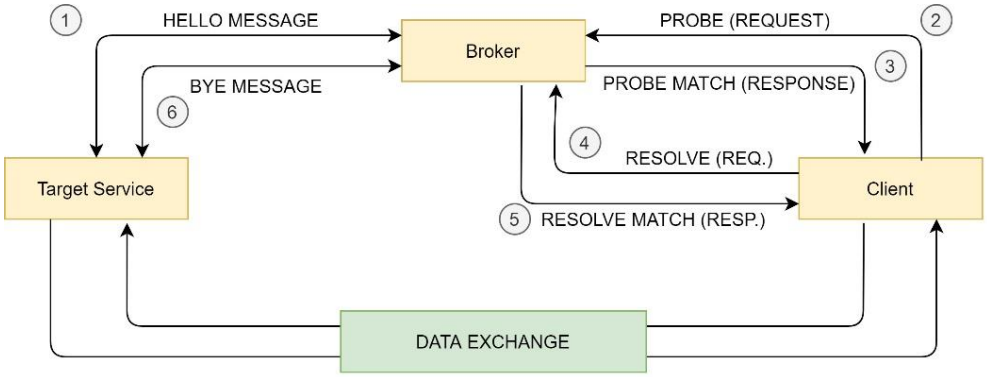

Figura 4. Descubrimiento de microservicios web: una vista general.

4.4.1. Descubrimiento

El bróker gestiona los procesos de publicación y descubrimiento de nuevos microservicios y facilita una capa de interoperabilidad con microservicios web externos. Está basado en WS-Discovery donde diferenciamos dos roles: los proveedores, que ofrecen datos, y los clientes subscriptores que los necesitan (consumidores).

La interoperabilidad con microservicios WS-Discovery nativos se podría implementar sobre un proxy que transformase fácilmente los mensajes nativos de WS- * en mensajes de coreografía. Por eso, los encabezados WS se han adaptado a los encabezados de mensajes de coreografía. En cuanto al cuerpo del mensaje, en este se incluyen explícitamente datos directos de los sensores y las medidas, pero no sobre el patrón de comunicación que soporta el proveedor. El bróker interpreta el patrón soportado a partir del texto del mensaje utilizando un intérprete basado en la clase Regexp[6].

El análisis del lenguaje natural con expresiones regulares (Regexp) nos permite reducir el cuerpo de los mensajes. Como un microservicio se puede diseñar para que acepte uno o varios patrones de intercambio de mensajes, para expresar tal diversidad solo con expresiones básicas el cuerpo del mensaje se construiría mediante una estructura anidada. Sin embargo, el uso de expresiones regulares permite expresar la diversidad en una sola cadena.

Cada nuevo proveedor que se conecta al sistema publica su "provisión" al microservicio bróker. Esta publicación se realiza con un mensaje de HELLO. Los metadatos del mensaje contienen el identificador de la dirección del microservicio y el cuerpo del mensaje incluye el tipo de sensor y medición y la frecuencia de recopilación.

Por otra parte, cuando un consumidor se conecta al sistema, pregunta al bróker con un mensaje de PROBE qué microservicio le puede proveer de unos datos determinados: tipo sensor y medición. El bróker verifica si hay algún proveedor que cumpla con los requisitos solicitados y resuelve la solicitud con un nulo o con el identificador necesario para establecer la comunicación en un PROBE-MATCH. Los mensajes de RESOLVE sirven para que el consumidor elija en caso de que el bróker le ofrezca más de un proveedor. El RESOLVE-MATCH se trata como un mensaje de confirmación.

Los mensajes donde el cuerpo del mismo se expresa en lenguaje natural son los de HELLO y PROBE por ser los más complejos. Los mensajes de PROBE MATCH y RESOLVE MATCH se implementan con una notación en strings conocida por el receptor ya que se asume que sólo

[6] https://docs.microsoft.com/es-es/dotnet/api/system.text.regularexpressions.regex?view=net-5.0

el bróker implementa el uso de Regexp; el resto de los microservicios no. Esto es debido al consumo de CPU y complejidad que supone la interpretación de lenguaje natural.

Aunque en la sección de Implementación se verá con mayor detalle la manera en la que se produce el descubrimiento, la Figura 6 es un ejemplo de descubrimiento donde se observa con claridad que en este proceso sólo interactúan dos microservicios, un microservicio consumidor (que desea descubrir otro microservicio) y el bróker:

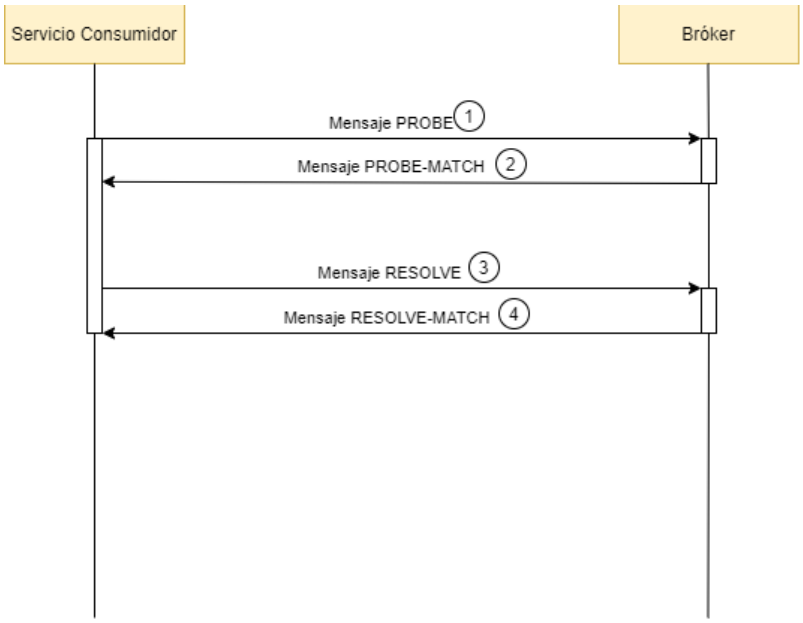

Figura 5. Esquema de descubrimiento de microservicios.

El descubrimiento es necesario en todos los patrones de comunicación que hemos definido:

- En el caso de REQ-RES, el consumidor necesita conocer el identificador del microservicio al que tiene que solicitar la información.
- En el caso DATACENTRIC, el consumidor necesita descubrir el identificador de los mensajes que le interesa recibir.
- El caso de INTELIGENTE, el consumidor necesita conocer el identificador de los mensajes que le interesa recibir, pero también el proveedor tiene que conocer la regla a configurar para ese consumidor. Esta configuración es parte del proceso de subscripción.

4.4.1.1. Patrón REQ-RES

La suscripción de solicitud–respuesta se trata de un patrón que se utiliza cuando un microservicio requiere una respuesta muy específica e inmediata de otro microservicio. Un proveedor de microservicios que se publica ofreciendo el modelo solicitud-respuesta atiende las solicitudes que se dirigen específicamente a él.

Una vez que el consumidor tiene el identificador del proveedor, los mensajes se intercambian directamente entre el proveedor y el consumidor.

Un proveedor puede responder con el último valor de sensor recibido en el dispositivo local o con un conjunto de datos entre dos marcas de tiempo. Para proporcionar un conjunto de datos, el proveedor debe solicitar datos al microservicio de base de datos antes de responder al consumidor. Por lo tanto, una solicitud-respuesta consumidor-proveedor puede implicar una cadena de solicitud-respuesta en el sistema de coreografía.

4.4.1.2. Patrón DATACENTRIC

La suscripción está basada en eventos asincrónicos. Un microservicio centrado en los datos transmite de forma asincrónica paquetes de datos recibidos de un sensor o conjunto de sensores. Como ventaja, se pueden implementar filtros, detección de errores y mecanismos de corrección en este microservicio. Después de recibir datos de uno o varios sensores, el proveedor procesa los datos y los transmite como un evento asincrónico sin esperar respuesta de los receptores. Los paquetes se identifican por los datos contenidos. Por tanto, varios consumidores pueden recibir un mismo paquete.
Además, un proveedor de este tipo no envía datos si no hay algún consumidor esperando. Así el tráfico de la red se reduce mediante la activación selectiva de estos microservicios. Para ello es necesario implementar un mecanismo que permita activar o desactivar el microservicio que envía datos cuando no exista ningún microservicio suscrito a él. Esto se hace por medio del bróker. El bróker mantiene una lista de clientes consumidores subscritos por proveedor. El algoritmo se describe cómo se produce el evento de ACTIVACIÓN de un microservicio proveedor en el momento que un consumidor se subscriba. En caso de que la lista se encuentre vacía, el bróker notifica al proveedor la NO ACTIVACIÓN. La Figura 6 muestra el concepto de ACTIVACIÓN.

Algoritmo 1. Suscripción y cancelar la suscripción de actualizaciones.
En eso: El bróker recibe un mensaje
CASE:
1. Recibe un mensaje HELLO de $target_i$; entonces $target_{i\_lista} = 0$
2. Recibe una suscripción a $target_i$; entonces $target_{i\_lista}$ ++ y envía un mensaje de ACTIVACIÓN
3. Recibe una cancelación de suscripción a $target_i$; entonces
   a. $Target_{i\_lista}$--;
   b. if list = = 0 entonces envía un mensaje de NO ACTIVACIÓN

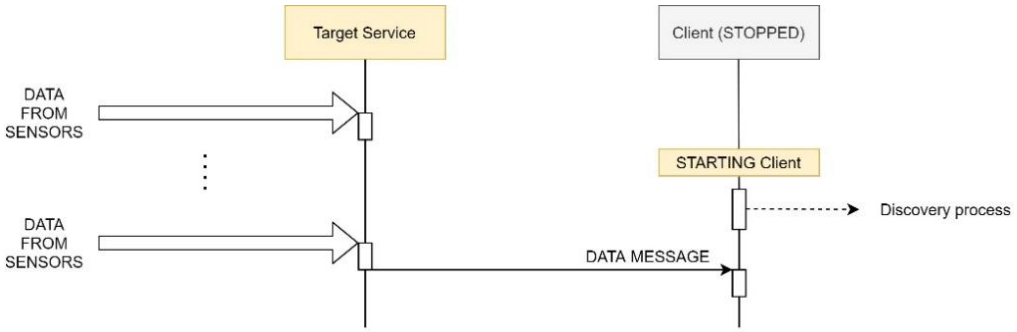

Figura 6. Patrón DATACENTRIC: activación selectiva.

4.4.1.3. Patrón INTELIGENTE

El patrón de un microservicio de reglas se basa en su capacidad de enviar eventos cuando se cumple una regla específica. En la Figura 7 se muestran algunos ejemplos de tipos de reglas: detección de un valor fuera de una tendencia establecida, obtención del promedio o detección de valores fuera de los umbrales y detección del valor máximo o mínimo entre dos marcas de tiempo. En el TFM se ha implementado como ejemplo la regla de “threshold”.

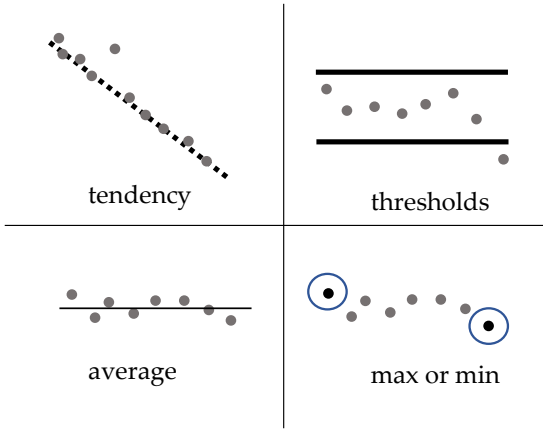

Figura 7. Ejemplos de reglas.

Un proveedor publica una regla o reglas que acepta en el bróker, por ejemplo, una regla de umbral de humedad inferior. Una regla puede implicar una o varias medidas. Un consumidor se suscribe a la regla indicando el valor var% correspondiente al umbral. Si la humedad cae por debajo del umbral impuesto por el consumidor, el proveedor enviará un mensaje a ese consumidor con el dato de humedad.

4.4.2. Publicación y anulación de publicación

Las herramientas que el bróker utiliza para realizar los procesos de publicación y anulación de publicación son los mensajes HELLO y BYE, una lista de microservicios publicados y varias listas de clientes suscritos por microservicio proveedor de datos. Los publicadores o proveedores de datos y el bróker intercambian información usando el patrón de solicitud-respuesta. Los mensajes HELLO y BYE son paquetes de solicitud. La respuesta esperada es un reconocimiento (ACK) que indica al microservicio el estado del proceso realizado.

Se presupone que el campo "scope" contenido en el cuerpo del mensaje de tipo HELLO se encuentra expresado en lenguaje natural, con expresiones regulares. El campo "scope" está especificado en el estándar WS-Discovery.

Un proveedor de datos puede soportar uno o varios modelos de comunicación a la vez. Por lo tanto, cada nuevo proveedor que se conecte al sistema debe anunciar al bróker los modelos de datos que ofrece y qué reglas acepta en caso de utilizar el modelo INTELIGENTE. El bróker, por su parte, mantiene una lista actualizada de los microservicios que se han publicado.

En el caso del modelo DATACENTRIC, el microservicio proveedor de datos se mantiene en espera y no envía datos mientras no haya ninguna suscripción. Tan pronto como el bróker detecta la primera suscripción de un cliente a este proveedor, el bróker se encargará de enviarle un mensaje para que comience a enviar datos al microservicio solicitante.

De manera similar, los mensajes de cancelación de suscripción actualizan las listas de clientes suscritos, provocando un evento de notificación a cada publicador en caso de que sean DATACENTRIC cuando la lista está vacía (algoritmo).

Es importante recalcar que los mensajes BYE actualizan la lista de publicación del bróker. Este tipo de mensajes es crítico para los clientes que utilicen el patrón INTELIGENTE. En el caso de que un microservicio proveedor de datos que utilice los patrones REQ-RES o DATACENTRIC deje de emitir datos, no es necesario que se avise al bróker de su finalización; los consumidores detectarán la ausencia de mensajes. Sin embargo, en el patrón INTELIGENTE es necesario que el proveedor avise al bróker indicando que no está disponible debido a que

podría ser que el microservicio solicitante no recibiese datos debido a que la regla no se cumple. Por lo tanto, es importante que el cliente sepa si el microservicio proveedor está activo o no.

El algoritmo de descubrimiento necesita dos pares de mensajes de tipo REQ-RES: PROBE y PROBE MATCH, y RESOLVE y RESOLVE MATCH, como se muestra en la Figura 5. En este algoritmo interactúan dos microservicios, el microservicio solicitante de datos o consumidor y el bróker, que es quien tiene la lista de los microservicios proveedores de datos publicados:

- PROBE (solicitud): El primer mensaje que envía el microservicio solicitante al bróker. El cuerpo del mensaje está comprendido el tipo de datos que necesita, la frecuencia mínima que desea obtener y la forma en la que prefiere recibir los datos.

- PROBE MATCH (respuesta): Mensaje de respuesta del bróker que anida tantas secciones <d: ProbeMatches> como coincidencias encontradas entre los microservicios publicados. Cada sección <d: ProbeMatches> incluye en el alcance el tipo de datos que ofrece, un patrón de entrega y la frecuencia con la que ofrece los datos, en caso de ser un microservicio que admita envíos periódicos. Los mensajes están formados por cadenas, lo que facilita el procesamiento por parte del cliente. Para reducir el tamaño, el bróker no necesitaría ofertar todos los proveedores, puede realizar una selección priorizando el DATACENTRIC que es el patrón que menos tráfico de red implica.

- RESOLVE (solicitud): El microservicio solicitante devuelve la selección. En caso de que no haya ninguna entrada válida, el microservicio consumidor no iniciará el segundo par de mensajes RESOLVE y RESOLVE-MATCH. En el caso de tratarse de un patrón INTELIGENTE, el mensaje también incluye la variable %var que es el valor con el que se aplicará la regla. Después de recibir un paquete RESOLVE, el bróker se comunica con el proveedor de microservicios de destino, informando la regla y la variable seleccionadas. Esta comunicación entre el corredor y el objetivo sigue un patrón de solicitud–respuesta.

- RESOLVE MATCH (respuesta): En el caso del patrón INTELIGENTE, los mensajes RESOLVE MATCH y RESOLVE significan lo mismo, y sólo se utilizan a modo de reconocimiento.

4.5. Interpretación de mensajes (Regexp)

Como ya sabemos, la comunicación entre microservicios es posible gracias a la transmisión de mensajes estructurados en el formato FIPA[7]. Es muy importante que los microservicios sepan interpretar correctamente los mensajes que se transmiten en nuestro proyecto dado que nos encontramos ante un sistema distribuido y por ello es necesario que las distintas partes que lo componen se comuniquen coherentemente entre sí.

En este punto se establece como objetivo principal la necesidad de un sistema que permita la comunicación de las distintas partes manteniendo un nivel sencillo de complejidad estructural, pero sin excluir gran parte de todo lo que se puede hacer utilizando sistemas de comunicación más complejos. Por ello hemos optado por añadir a nuestro sistema un modelo basado en la interpretación de mensajes utilizando expresiones simples. Para hacer esto posible, existe una pieza dentro del sistema que se encargará de realizar este análisis; esta parte del sistema en concreto está compuesta por un algoritmo de análisis de expresiones simples, aunque bastante complejo, pero que permite utilizar unas reglas de comunicación muy ‘relajadas’. Así se pretende eliminar la necesidad de conocer un protocolo de comunicación común con el objetivo de facilitar el acoplamiento de elementos en el futuro o la interoperabilidad con nuevos sistemas.

[7] http://www.fipa.org/specs/fipa00071/XC00071C.html

El uso de expresiones regulares en el campo del mensaje permite que los microservicios se comuniquen ‘entre sí’ utilizando lenguaje natural. Esto implica una mayor facilidad (y rapidez) a la hora de crear un mensaje, debido a que no se requiere un consenso sobre una estructura común o una especificación concreta, aunque esto no es exactamente así para todos los casos. Esto se debe a que los mensajes en el entorno de coreografía tienen un formato estructurado que está siempre en la línea de la especificación SOAP. Es necesario que la cabecera de los mensajes contenga información contextual acerca de quién es el emisor, quién es el receptor y qué máscaras de recepción tiene, etc., y esta información siempre estará estructurada. Aun así, se permite que en el contenido o cuerpo del mensaje los microservicios expresen sus necesidades mediante dos configuraciones diferentes:

- En primer lugar, dos microservicios se pueden comunicar utilizando una estructura común a ambos, que es conocida por todos los microservicios que forman el sistema. Esta configuración es necesaria para microservicios que carezcan de la capacidad de hacer peticiones en lenguaje natural.
- La segunda configuración habilita que los microservicios expresen sus necesidades en lenguaje natural y no siguiendo una estructura específica común.

De esta forma se obtiene un sistema versátil y con mayor flexibilidad frente a los sistemas tradicionales debido a que los microservicios pueden comunicar sus datos de distinta forma, según convenga.

Los microservicios pueden realizar el proceso de descubrimiento utilizando ambos mecanismos, ya sea en lenguaje natural o estructurando la petición en el cuerpo del mensaje, siguiendo la especificación. Ahora bien, no es interesante aplicar la función de entender ambos tipos de configuraciones a todos los microservicios puesto que existen interacciones que sólo tienen como finalidad transmitir datos que ya poseen una estructura definida. En ese caso hemos considerado que no es necesario transformar esta estructura a lenguaje natural para transmitir la información, proceso que exige tiempo y recursos. Sin embargo, estimamos oportuno la necesidad de aceptar este tipo de mensajes en el proceso de descubrimiento y publicación del bróker, ya que entendemos que es en ese proceso cuando se produce un mayor intercambio de información, que puede ser expresada utilizando lenguaje natural y que ocuparía mucho más tamaño si se tuviese que estructurar. De esta forma conseguimos reducir el tamaño final del mensaje y, en consecuencia, se mejora la eficiencia general del sistema. Además, con la implantación del análisis de expresiones simples se aporta una mayor resistencia a posibles fallos o errores en el formato del mensaje.

En nuestro caso, el sistema real posee varias interacciones en lenguaje natural y otras muchas en lenguaje estructurado. En concreto, el único microservicio que necesariamente debe admitir peticiones utilizando ambas configuraciones es el bróker. Si esto no fuese así y sólo admitiese el patrón estructurado, los microservicios que sólo se comunicasen mediante lenguaje natural estarían excluidos del proceso de descubrimiento y publicación de microservicios -proceso que está manejado por el bróker- y esto no nos interesa.

Aun así, hay que tener en cuenta que para hacer posible el uso de dos configuraciones distintas en el campo del mensaje es necesario que el agente receptor del mensaje sepa discernir entre ambas configuraciones disponibles. En el caso de que el receptor del mensaje únicamente acepte mensajes estructurados, si recibe un mensaje que contiene información en lenguaje natural, desechará el mensaje y seguirá con su ejecución normal. No hay más que realizar un pequeño análisis para diferenciar entre un mensaje estructurado y otro en lenguaje natural. Se trata de un proceso que pueden realizar todos los microservicios sin mayor complejidad; sin embargo, para que el receptor sea capaz de obtener la información contenida en un mensaje formado con lenguaje natural es necesario que utilice las funciones definidas para ello. Estas funciones tienen como objetivo el análisis y extracción de los datos necesarios a través del análisis Regexp. Los microservicios que deseen utilizar la configuración basada en lenguaje

natural deberán utilizar esta implementación para poder obtener la información contenida en el mensaje.

Si nos centramos en la implementación del análisis de mensajes en lenguaje natural mediante Regexp, es importante mencionar que, en nuestro caso, se trata de un análisis que busca ciertos patrones en el campo 'scopes' del mensaje y que, en caso de encontrarlos, extrae los datos importantes para el sistema, entre los que se incluyen el tipo de datos que envía un microservicio, la frecuencia de emisión de datos o el modelo del sensor, entre otros. La implementación que realiza la búsqueda de patrones utiliza como base las librerías de referencia para el análisis de Expresiones Regulares (Regexp) en C#[8]. Además, ha sido preciso tener en cuenta la necesidad de distinguir entre los tipos de microservicio de datos al que pertenece el microservicio que compone el mensaje a partir de su contenido en lenguaje natural. Por ejemplo: expresiones como '*cada X tiempo*' sirven para indicar que existe una cierta periodicidad entre envíos de mensajes, por lo que nuestro algoritmo determinará que el microservicio que ha emitido ese mensaje es de tipo DATACENTRIC, ya que es el único patrón que ofrece datos a partir de una frecuencia dada. Para ello, el algoritmo divide el contenido del mensaje en expresiones simples en los lugares donde existe una conjunción gramatical, de tal forma que el contenido de un mensaje queda dividido en distintas oraciones que deben ser analizadas por separado, y cada una corresponde con una descripción del microservicio o 'ofrecimiento'. Como se comentaba previamente, esta información debe estar contenida en el campo 'Scopes' del mensaje. Por ejemplo, un microservicio que desee publicarse en el bróker utilizando lenguaje natural podría hacerlo componiendo el campo 'Scopes' del mensaje de la siguiente forma:

"Ofrezco datos de humedad cada 30 minutos en el canal 0 ó atiendo peticiones de tipo REQ-RES."

El anterior ejemplo muestra el mensaje de un microservicio que ofrece dos tipos de datos, DATACENTRIC y REQ-RES. El algoritmo de análisis divide el mensaje por su conjunción, en este caso un 'o', y analiza las oraciones por separado. Después, cada oración se analiza de forma individual para asignar al microservicio emisor de esa oración un tipo de datos concreto. La primera oración, "*Ofrezco datos de humedad cada 30 minutos en el canal 0*", corresponde con un microservicio de tipo DATACENTRIC, debido a que tiene una expresión de periodicidad. Sin embargo, la segunda oración, que se corresponde con "*atiendo peticiones de tipo REQ-RES*." determina que el microservicio también es de tipo REQ-RES ya que atiende peticiones, según se indica en el mensaje. De esta forma el bróker podrá asignar un tipo al microservicio y realizar correctamente el proceso de descubrimiento, por lo que ya es compatible con lenguaje natural. Posteriormente se mostrará con detalle el funcionamiento de este algoritmo.

Como se comentaba anteriormente, es necesario que los microservicios que deseen comunicarse utilizando la configuración basada en el lenguaje natural posean la implementación descrita. Por ello se ha creado una librería que contiene las funciones necesarias para realizar este análisis de tal manera que cualquier microservicio pueda utilizarlas. Por otro lado, se ha realizado un caso de estudio para mejorar la comprensión de las ventajas que ofrece la configuración basada en el lenguaje natural frente a la configuración que ofrece una estructura en el cuerpo del mensaje. El análisis está enfocado a la eficiencia de las comunicaciones y procesado de la información y se ubica en la sección 5, relacionada con la implantación de este documento.

[8] https://docs.microsoft.com/es-es/dotnet/api/system.text.regularexpressions.regex?view=net-5.0

4.6. Visualización de resultados

En los últimos diez años se ha observado un crecimiento exponencial en la tecnología móvil hasta tal punto que la capacidad actual de los móviles se sitúa muy cerca de los equipos de sobremesa. Este fenómeno ha ido acompañado de constantes avances en las tecnologías que se utilizan en estos dispositivos. Estas tecnologías comprenden desde el hardware utilizado, como la cámara, la memoria RAM, el sensor de huellas, etc. hasta el software, que también ha sufrido muchos cambios. Actualmente existen entornos de desarrollo muy completos que dejan a disposición del programador todas las herramientas que contiene un dispositivo móvil y que permiten implementar casi cualquier solución. Generalmente, la gran mayoría de dispositivos móviles, tal y como los conocemos, utilizan sistemas operativos bien conocidos, como Android o Apple, por lo que desarrollar una aplicación para uno de esos sistemas operativos garantiza que gran parte de los dispositivos actuales puedan ejecutar dicha aplicación. Además, recientemente ha surgido un nuevo paradigma de programación que se denomina 'cross-platform' y que permite desarrollar aplicaciones ejecutables en distintos sistemas operativos y, en consecuencia, facilita el uso de la aplicación por un mayor número de usuarios. Tiene la ventaja añadida de tener que desarrollar sólo un proyecto único, y no un proyecto individual para cada sistema operativo. El hecho de usar 'cross-platform' para desarrollar la aplicación ofrece muchas más ventajas que inconvenientes, tales como una mayor rentabilidad del desarrollo frente a desarrollos tradicionales y una reducción de carga de trabajo. Aun así, las aplicaciones de este tipo no tienen el mismo rendimiento que una aplicación expresamente diseñada para un sistema operativo concreto.

4.6.1. Análisis de la aplicación

En nuestro proyecto, ofrecemos una solución que abarca las dos tecnologías anteriormente descritas. En concreto, se ha implementado una aplicación de tipo 'cross-platform' y el sistema de coreografía conectado a la aplicación. Esta aplicación recibe los datos de nuestro sistema distribuido para mostrarlos en gráficas y ofrecer al usuario una visión global del estado del sistema. Para que esto sea posible se ha utilizado un agente intermedio entre la aplicación móvil y el sistema de coreografía. Esta pieza se encarga de almacenar los datos útiles para el usuario en nube para que la aplicación los pueda utilizar en el futuro. En nuestro caso, el agente intermediario es un servidor de bases de bases de datos: Firebase.

La aplicación implementada dentro del proyecto eSGarden está diseñada para observar valores de sensores provenientes de huertos escolares. La toma de decisiones sobre riego automático (u otros controles) recae en la capa Edge. El usuario se despreocupa. Por tanto, una app asociada sirve para visualizar datos y tomar decisiones manuales en base a alertas.

Para poder entender bien los beneficios que aporta el uso de estas tecnologías, se ha planteado una estructura global (todos los elementos del proyecto la conocen) en la que sus elementos superiores son los huertos, y cada huerto contiene varias parcelas. Las parcelas, a su vez, agrupan las mediciones de los sensores individualmente, y distinguen entre los tipos de sensores conectados en cada parcela. De este modo se puede visualizar el histórico de un tipo de datos enviado por cualquiera de los sensores que se sitúan en una parcela, como los de humedad y temperatura, por ejemplo. Además, existen espacios generales asociados a estaciones de meteorología, invernaderos o zonas de compostaje.

Para mantener esta organización, es necesario realizar el despliegue de los sensores de forma adecuada. El proceso de etiquetado de los sensores es crítico, debido a que es aquí donde se crean las relaciones entre los sensores, las parcelas y el huerto. Todos los sensores de una misma parcela deben ir etiquetados con el mismo número de parcela y el mismo número de huerto. Además, en la documentación de la aplicación se presenta una lista que relaciona los tipos de datos existentes (humedad, temperatura, luminosidad, CO2…) con el código que se

debe poner en el sensor para que el sistema identifique cada tipo de datos. Aun así, el proceso de registro de parcelas y huertos, así como el almacenaje de los datos recae únicamente en el sistema distribuido, siendo este el único responsable de desplegar los elementos necesarios para que la aplicación funcione. Así obtenemos un sistema distribuido en niveles lógicos, lo que facilita mucho el tratamiento de los datos.

Finalmente, es necesario que en el sistema implementado exista una parte encargada de comunicarse con el servidor de bases de datos externo con tal de transmitir la información que necesita la aplicación. Con ese fin se ha creado un microservicio que se conecta con Firebase y que realiza las peticiones necesarias para reunir todos los datos posibles y enviarlos a la base de datos externa. Este microservicio almacena la información en formato JSON en las secciones correspondientes al tipo de datos en el servidor. Para ello, el servidor debe de tener desplegada una pequeña infraestructura que permita al microservicio recolector de datos, dejar esos datos en la parcela idónea. Además de la creación de la estructura de huertos-parcelas-sensores, que es necesaria para el correcto funcionamiento de la app, también se ha añadido una sección paralela que permite la gestión de verduras y hortalizas dentro de la app. De esta manera, los usuarios pueden asignar un vegetal a una parcela y obtener información relacionada con ese vegetal, además de ayudas, tutoriales y alertas personalizadas. Por ejemplo, el usuario puede establecer alertas que detecten cuándo un valor de humedad supera o disminuye de un límite, que se introducirá manualmente por el usuario. Ese umbral puede establecerse a partir de los datos que la propia aplicación ofrece debido a que sabe qué vegetal hay en cada parcela.

El desarrollo de la solución móvil ha necesitado una estructura bien definida en la base de datos. Esa estructura, que está expuesta en el servidor Firebase, relaciona las distintas parcelas y huertos manteniendo la lógica del sistema real, por medio del lenguaje JSON (JavaScript Object notation). Este lenguaje se utiliza para estructurar el contenido de nuestra base de datos, y típicamente se usa para intercambiar información en texto legible por el usuario, es decir, en texto plano. La información se modela mediante la estructuración y anidación de etiquetas y datos. Básicamente el lenguaje se basa en propiedades que tienen un valor (que puede ser una o varias propiedades o un valor). A continuación, se muestra un ejemplo de información real modelada en JSON:

```
{
  "huerto":8,
  "nombre":"Jardin del colegio 1",
  "ubicacion": "Valencia",
  "datos":[
    {
      "tipo":"Humedad",
      "dato":"45%"
    },{
      "tipo":"Temperatura",
      "dato":"25°"
    }
  ]
}
```

El desarrollo de aplicaciones Android últimamente se enfoca en este tipo de lenguajes de texto plano para estructurar su contenido debido a que no requiere de librerías complejas ni conocimiento avanzado en bases de datos para el correcto almacenamiento de estos. Existen muchos proyectos que utilizan este tipo de lenguajes (JSON, XML, etc.) y lo explotan al máximo en tecnologías móviles y web (como Javascript, que integra JSON para definir objetos) debido a que se trata de una tecnología novedosa y que aporta ventajas frente al almacenamiento de grandes cantidades de datos en bases de datos relacionales. Por ese motivo se ha optado por el lenguaje estructurado, debido a que se trabajará con mucha información, entre otras razones.

Para entender mejor el problema, es útil entender la estructura de la base de datos como un grafo con estructura de carpetas y archivos, en la que las carpetas pueden contener archivos y

carpetas y los archivos son los elementos 'hoja' de la estructura. A continuación, se presenta el modelo ontológico que se ha seguido para estructurar la base de datos. Se trata de un diseño en el que cada elemento de tipo hoja equivale a un par clave-valor sin elementos en su interior, y un elemento de tipo carpeta puede contener más elementos en su interior, ya sean hojas, carpetas o ambos:

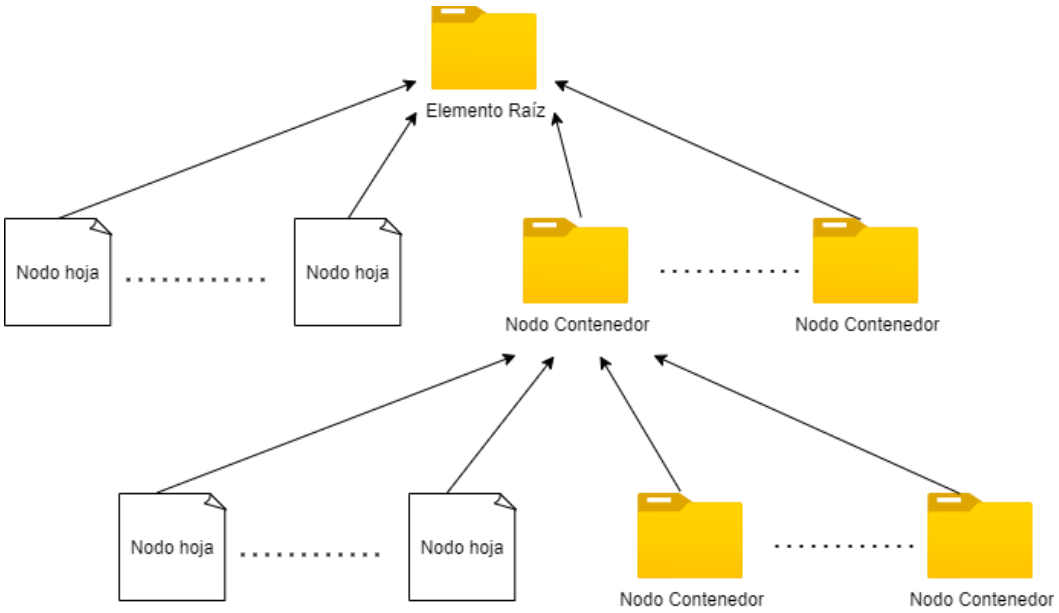

Figura 8. Modelo ontológico de la estructuración realizada en el servidor de Bases de datos

La estructuración de la base de datos se ha hecho considerando varios consejos a la hora de diseñar la lógica de la aplicación. Por ejemplo, las siguientes premisas para el diseño de la base de datos consideramos que son de gran importancia para facilitar su mantenimiento:

4. Un número muy elevado de niveles dentro de la estructura JSON de la base de datos ralentiza el funcionamiento del sistema y dificulta la búsqueda de información. En el caso de que la estructura crezca con el tiempo, nos sirve de gran ayuda utilizar un diseño muy simple.
5. Es muy importante agrupar la información que está relacionada entre sí, aunque esta información esté 'por definición' en otra parte de la estructura lógica. Por ejemplo, si tenemos una serie de parcelas por cada huerto, podrá parecer lógico crear las parcelas dentro de la estructura del huerto. Sin embargo, si deseamos obtener los datos de una única parcela, deberemos obtener todo el huerto en el que ya están incluidas otras parcelas y procesar la información para obtener la parcela que queremos.

Aún teniendo en cuenta ambas consideraciones, habitualmente cuando se realizan peticiones de datos se suele solicitar el mismo tipo de datos, o el conjunto de todos los datos de un mismo tipo, o todos los datos de un huerto, etc. Por ello, es interesante agrupar la información según se utilice. Es decir, agrupar los datos de las parcelas en un nivel muy cercano a la raíz o incluso en la propia raíz y que tengan una propiedad que indique cuál es el huerto al que pertenecen.

5. Solución Propuesta

A continuación, se expone el proceso de desarrollo del sistema presentado en el anterior apartado. Se han implementado satisfactoriamente los objetivos establecidos en los análisis iniciales y el resultado final es una arquitectura IoT madura y con gran capacidad. En primer lugar, nos centraremos en aspectos generales de la solución tales como la planificación del trabajo, los presupuestos y costes del proyecto. En esta sección es donde se presentarán las distintas fases del proyecto y el tiempo que se ha utilizado en cada una. En segundo lugar, nos centraremos en varios aspectos del diseño que han servido para distinguir las características más importantes. Así se ofrece una visión detallada del sistema a nivel de funcionalidad y herramientas utilizadas.

5.1. Plan de trabajo

A continuación, se dan a conocer los procesos que han permitido completar el sistema en el tiempo establecido. También se presentan los recursos utilizados y se expone la forma de trabajar que se ha llevado a cabo durante el transcurso de este proyecto. Posteriormente se analizarán las distintas fases y los recursos utilizados en cada fase.

Se ha realizado una planificación inicial considerando el inconveniente que supone trabajar en un proyecto que implica una fase inicial de investigación. La planificación temporal puede variar mucho de lo inicialmente establecido debido a que no se cuenta con la experiencia suficiente sobre las tecnologías en estudio. Aun así, se ha intentado seguir la planificación de la forma más estricta posible y al final el resultado ha sido positivo; el tiempo consumido no excede mucho del tiempo planificado, y los plazos de entrega no se han tenido que rectificar, aunque esto último ha sido posible gracias a que se ha tenido en cuenta un periodo de margen para cada entregable.

Se ha aplicado desde los inicios del proyecto una metodología ágil que ha permitido estar siempre al día de los nuevos avances y añadir a la solución nuevas tecnologías durante su desarrollo. Como se trata de una metodología ágil, el desarrollo no ha seguido una estructura estricta, pero se le ha dado mayor importancia a obtener cada cierto tiempo un proyecto que se pueda ejecutar y probar. Estas iteraciones se han validado con la tutora del proyecto en las reuniones, que determina si la implementación se ha realizado correctamente. Además, el hecho de validar reiteradamente el desarrollo con la tutora cada poco tiempo ha facilitado la corrección prematura de errores que hubiesen sido muy graves en un futuro. De forma adicional, las reuniones no sólo han servido para detectar errores, sino que también han servido como base para realizar análisis en retrospectiva que han facilitado el avance del proyecto y han servido para extraer conclusiones más acertadas. Por otro lado, se ha dedicado más tiempo a la investigación y a realizar pruebas sobre el software que se iba realizando, lo que significa un gasto invisible en recursos para el proyecto. Esto ha servido para comenzar a validar el sistema desde etapas muy tempranas y ha permitido el desarrollo de las etapas posteriores sobre una implementación estable y probada.

En concreto se ha establecido un periodo semanal para realizar estas reuniones, que debido a la situación actual de pandemia se han llevado a cabo de forma telemática. Inicialmente, en las primeras reuniones se comenzó a analizar del problema planteado, proceso a partir del cual se estableció la estructura externa del sistema (qué componentes físicos lo forman y cómo se relacionan entre ellos). Una vez que estuvieron claras la disposición de las Raspberry pi que forman la capa Edge, la estructura de la base de datos y también la aplicación móvil de visualización, hubo que realizar varias aproximaciones de estructura a nivel de Software, analizando la interacción entre microservicios. Estas aproximaciones han servido para ofrecer una visión de los microservicios que componen el sistema, indistintamente de la ubicación física

donde se encuentran. Este proceso fue uno de los más costosos de completar en términos temporales debido a que la cantidad de elementos a tener en cuenta era enorme y las tecnologías podían no ser compatibles. Se tuvieron que llevar a cabo estudios de compatibilidad de prácticamente todas las tecnologías interconectadas para garantizar que no se produjese ningún problema a la hora de desplegar los microservicios. Además, se realizaron estudios sobre la cantidad de microservicios que puede ejecutar una Raspberry Pi con el objetivo de determinar el máximo número de microservicios que se pueden ubicar físicamente en cada una para poder hacer un diseño con mayor coherencia con el sistema real.

Cuando se obtuvo una organización válida de la estructura del sistema comenzó el proceso de implementación. Como se comentaba previamente, periódicamente en las reuniones, se han realizado varios análisis sobre la implementación realizada hasta el momento y se han extraído conclusiones que se tienen en cuenta para las futuras iteraciones. De esta forma hemos obtenido información que ha sido clave para el correcto funcionamiento del sistema en fases tempranas y no al final del desarrollo del proyecto, además de detectar problemas graves de conectividad y despliegue en los inicios del proyecto. En concreto las secciones que más cambios han sufrido desde el comienzo son dos secciones muy relacionadas entre sí: la sección de interacción entre microservicios y la capa de patrones de comunicación de datos. Estas dos partes del sistema son las partes más críticas y han estado sujetas a cambios importantes. Inicialmente se contaba con un pequeño conjunto de microservicios que realizaban muchas funciones internas, y posteriormente se determinó que era mejor seguir un modelo de más microservicios pero con menor y más clara funcionalidad.

Pero para determinar con mayor exactitud el gasto de recursos que ha supuesto cada uno de los procesos necesarios para llevar a cabo este proyecto, se presenta la siguiente división de fases que se ha seguido este proyecto.

- Aprendizaje inicial necesario para el desarrollo de aplicaciones en Visual Studio y #C. (Fecha inicio: 29 septiembre 2019 – Fecha fin:1 noviembre 2019)

- Entender y conocer en profundidad el motor de coreografía realizando algunas pruebas iniciales. (Fecha inicio: 1 noviembre 2019 – Fecha fin: 1 enero 2019)

- Aprendizaje en el desarrollo de aplicaciones orientadas a dispositivos móviles utilizando el IDE de Android Studio, Flutter y una interconexión con Firebase. (1 enero 2020 – 15 marzo 2020)

- Aprendizaje en programación de Arduinos para formar una sencilla red WSN. (Fecha inicio: 15 marzo 2020 – Fecha fin: 15 abril 2020)

- Desarrollo de las primeras versiones de microservicios y patrones de comunicación sobre el motor de coreografía. (15 abril 2020 – 1 junio 2020)

- Definición y programación de los microservicios de conexión TCP entre motores de coreografía. (1 junio 2020 – 15 junio 2020)

- Primeras pruebas de tiempos sobre la interacción de microservicios según los distintos patrones de comunicación. (15 junio 2020 – 30 junio 2020)

- Planteamiento, análisis y desarrollo del bróker y microservicio de descubrimiento. (30 junio 2020 – 30 septiembre 2020)

- Aprendizaje y aplicación de Regexp en proceso de análisis de mensajes. (1 agosto – 1 agosto 2020)

- Pruebas de descubrimiento con diferentes ejemplos y extracción de tiempos. (1 agosto 2020 – 1 septiembre 2020)

- Participación en dos artículos de investigación. (1 septiembre 2020 – 1 enero 2021)

- Planteamiento, análisis y desarrollo de la aplicación Android (en paralelo, 1 enero 2020 – 1 junio 2020)

- Pruebas de la app integrando datos enviados por los Arduino a Firebase. (1 febrero 2021 – 1 marzo 2021)

- Subida de la app a Google Play. (1 marzo 2021)[9]

Además de las horas de dedicación al TFM, he disfrutado de un contrato de trabajo a cargo del proyecto eSGarden: "School Gardens for Future Citizens", con el número de referencia: 2018-ES01-KA201-050599, financiado por el SEPIE en el programa Erasmus + de Asociaciones Escolares, y que me ha permitido invertir horas adicionales en el desarrollo de la aplicación móvil. Este contrato ha tenido una duración desde el 29 de septiembre de 2019 hasta el 29 de julio de 2020 y ha servido para aprender las tecnologías en estudio y desarrollar la aplicación móvil. Aunque en el contrato nos hemos centrado en la aplicación, también ha servido para investigar distintos tipos de sensores, pasarelas que reciben la información y tecnologías de comunicación de baja potencia y alto alcance.

5.2. Presupuesto

La obtención del presupuesto ha sido posible gracias a que se ha seguido una planificación y se han estimado apropiadamente los recursos necesarios para desarrollar el sistema. Para obtener el presupuesto total se han tenido en cuenta los costes del sistema en términos de hardware y software adquirido, el coste de almacenamiento en un servidor de bases de datos externo y el coste de los recursos humanos. Por otro lado, también se valoran las tecnologías investigadas que por algún motivo no se han adoptado en nuestro sistema pero que han supuesto algún tipo de gasto de recursos.

5.2.1. Recursos informáticos

A continuación, se determina el tipo y número de elementos hardware y software que se utilizan en el sistema. También se tienen en cuenta recursos que por algún motivo se han despreciado en la solución final, pero que han supuesto un coste económico en los inicios del proyecto. Para mostrar los distintos componentes que se han utilizado se presenta la siguiente tabla:

Hardware	Nº	Precio/unidad (Euros)	Precio total	¿Sirve?
Raspberry pi 3B +	3	40 €	120 €	Sí

[9] https://play.google.com/store/apps/details?id=myschool.esgarden&hl=en&gl=US

Ordenador	1	800 €	800 €	Sí
Arduino UNO Wifi rev.2	3	46 €	138 €	Sí
Arduino NANO (Imitación)	5	5 €	25€	Sí
Sensores de humedad	10	1 €	10	Sí
Sensores de temperatura	10	1 €	10	Sí
Sensores de luminosidad	5	1 €	5	Sí
Sensores de pluviometría	5	1 €	5	Sí
Sensores de sonidos	5	1 €	5	No
Sensores de calidad del aire	10	1 €	10	Sí
Sistema de coreografía	1	Cedido (0 €)	0	Sí
Arduino NANO IoT	5	16’6 €	83	No
Licencia SyncFusion	1	300 €	300	Sí
Servidor de base de datos	1	Gratuito	0	Sí

5.2.2. Recursos humanos

Este proyecto se ha desarrollado enteramente por una persona. Se ha dedicado un total de 360H que corresponden a los créditos del TFM, periodo que tiene una duración de 3,5 meses. También se han dedicado 300H provenientes del contrato de trabajo en la UPV para el desarrollo de la aplicación móvil, que, aunque no estaba pensada para este fin, se ha podido adaptar al sistema sin problemas y por lo tanto se ha podido reutilizar el trabajo realizado. Así también se tiene en cuenta el tiempo dedicado a la aplicación en nuestro proyecto ya que también forma parte de él.

Para obtener el salario total se ha aproximado un coste por hora similar al ofrecido en el contrato de trabajo que se corresponde con un salario aproximado de 10 E / Hora. A continuación, se presenta una tabla con el coste temporal de cada etapa.

Fase	Tiempo total	Coste por hora	Coste total
Sistema distribuido de Coreografía	360H	10’4H	3744 Euros
Aplicación Móvil	300H	10’4H	3120 Euros

A partir de esta tabla obtenemos que los costes relacionados con el salario de un empleado son de 3.744 Euros + 3.120 Euros, que en total suman 6.864 Euros. Además, se ha debido realizar un pago a la seguridad social relacionado con la cuota patronal del contrato de trabajo. Este importe es de 2.776 Euros, pero como sólo se han dedicado 300H del contrato a la implementación del TFM, que es un 40% del tiempo total, el importe que se ha pagado por la cuota patronal es de 910 Euros (el 40% de 2.276 Euros). En este punto se puede obtener una aproximación salarial pero no se tendrían en cuenta los costes adicionales.

En total la suma de recursos informáticos y humanos es de 9285€. Esta cifra proviene de la suma de 1511€ (recursos informáticos), 6864€ (recursos humanos) y 910€ (cuota a la patronal).

5.3. Arquitectura del sistema

En el siguiente apartado se expone estructura de nuestro sistema. Nuestro objetivo es primero presentar el sistema desde una visión externa para facilitar su entendimiento y posteriormente profundizar en cada una de las partes que lo forman. Por este motivo planteamos una división en la arquitectura; por un lado, presentamos la estructura externa de los componentes del sistema y por otro lado realizamos un diseño con gran nivel de detalle que determine cuales son y qué funciones tienen cada una de las partes existentes.

5.3.1. Grafo explicativo

La primera de las aproximaciones se corresponde con un modelo de elevada granularidad, es decir, con muy poco nivel de detalle. En él se exponen los distintos elementos físicos que interactúan en nuestro sistema y cómo se relacionan entre sí ya que es importante entender este modelo para profundizar más en el análisis. En la primera imagen se muestra la infraestructura de componentes que forman el sistema sin dar detalles acerca de su funcionamiento interno:

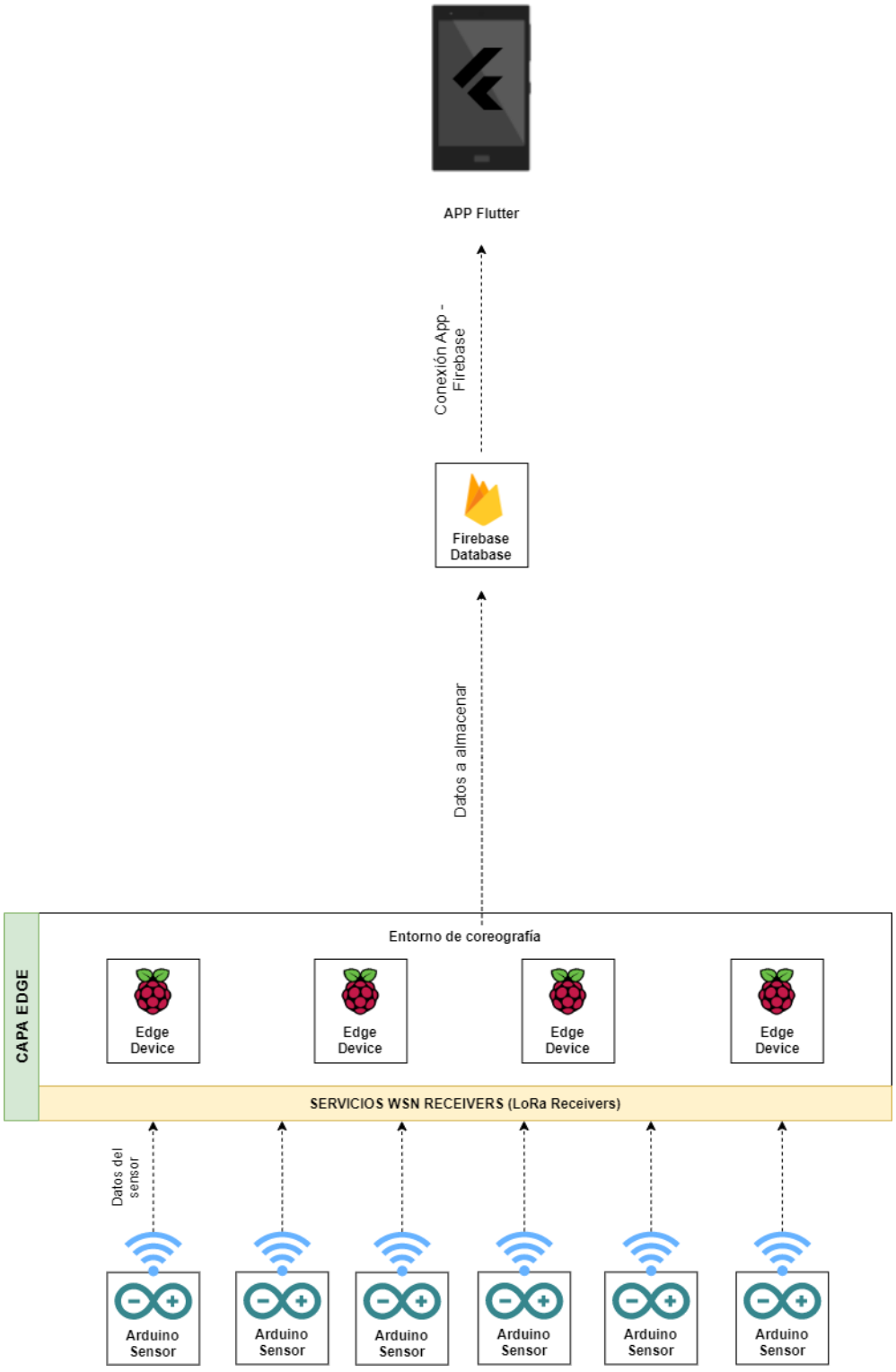

Figura 9. Arquitectura del sistema

En el esquema anterior existen cuatro secciones ordenadas en sentido descendente, siendo la capa más elevada la más próxima al usuario. El nivel más bajo de todos es el nivel de origen de la información. En este nivel inferior se lleva a cabo el proceso de recogida de los datos, proceso que se encargará de administrar la red de sensores (WSN), recoger la información y enviarla a la posterior capa. La capa receptora de estos datos es la más compleja, pues aquí se encuentran tanto la implementación de la tecnología necesaria para crear el entorno de coreografía como el sistema distribuido de microservicios en sí. En esta sección se pone en

práctica el concepto de computación Edge, ya que se trata de una capa distribuida entre dispositivos que pueden procesar la información que reciben y actuar sobre la red de sensores sin necesidad de que la información fluya hasta capas superiores. De esta manera las latencias y el número de peticiones a la base de datos central se reduce en comparación con los sistemas tradicionales. La información en esta capa se traspasa de manera organizada a la base de datos por medio de un microservicio cuya función es exactamente la de enviar los datos que se reciben localmente en cada uno de los dispositivos que forman la capa Edge. Para ello, también es necesario que en cada dispositivo exista una pequeña base de datos que almacene la información a enviar para solucionar problemas derivados de la falta de conectividad. Posteriormente, si el microservicio consigue conectarse a internet, envía esta información hacia el servidor de Firebase. Este servidor será el backend de la aplicación, que se corresponde con la sección superior en el esquema. La app realizará las consultas necesarias contra el servidor Firebase para mostrar al usuario final la información recogida por los sensores, pero sólo permite la visualización de la información y no ofrece funciones de actuación sobre los datos.

5.4. Diseño detallado

La siguiente aproximación pretende explicar la estructura a un nivel mayor de detalle y su objetivo principal es dar a entender el funcionamiento de la sección Edge, pues consideramos que es aquí donde se encuentra el punto de interés de nuestra solución. En esta capa coexisten las máquinas Raspberry pi, aunque es posible formar un sistema con una única Raspberry pi sin que ocurra ningún problema. De esta forma se garantiza la existencia un sistema con una alta cohesión, pero muy bajo acoplamiento.

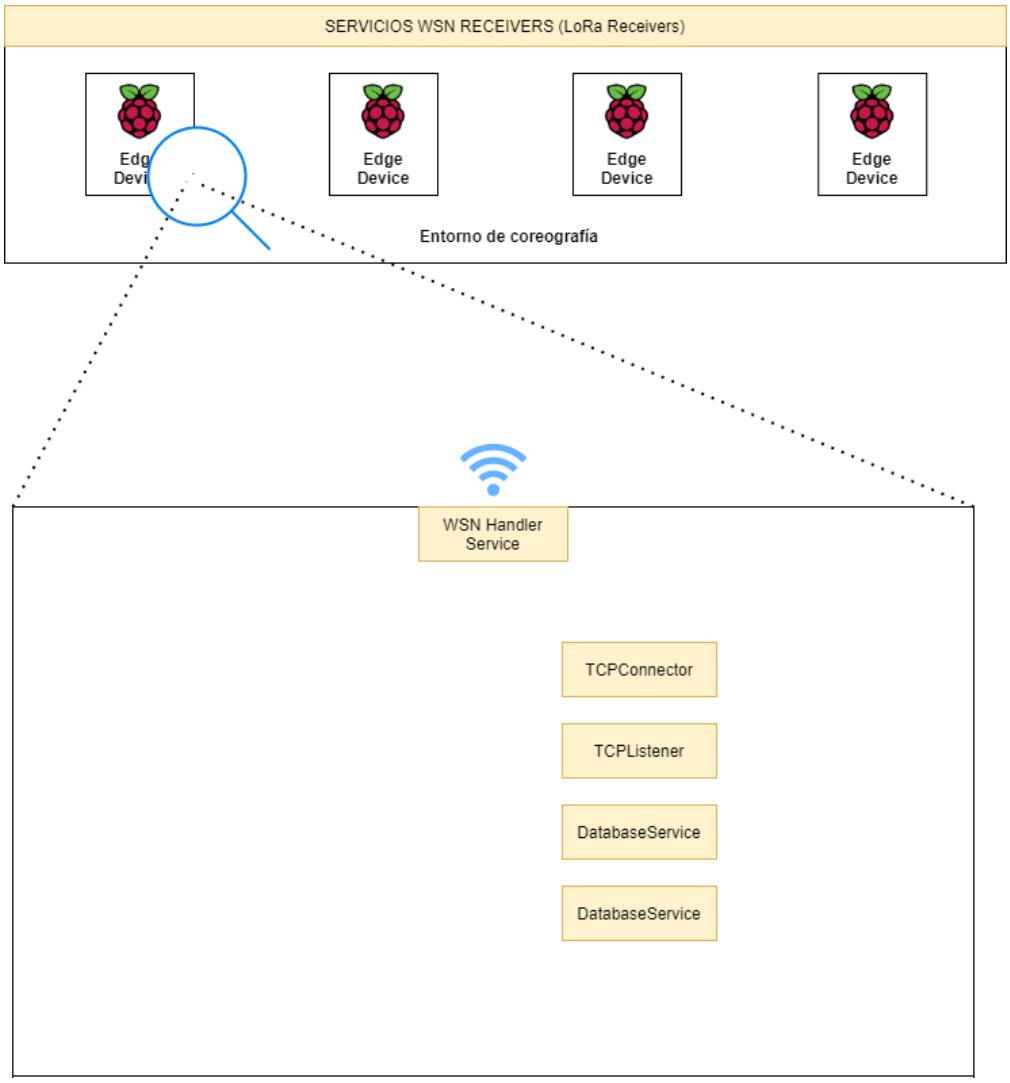

Figura 10. Arquitectura interna de una única Rasperry pi

En la figura anterior se observa la sección que corresponde a la capa Edge de nuestro sistema. Esta capa está formada por varias Raspberry pi, como se ha mencionado anteriormente, que son los dispositivos Edge en el diagrama. Cada dispositivo funciona de manera independiente y no necesita que haya más dispositivos para poder funcionar. Aun así, el sistema permite que los dispositivos se comuniquen entre sí para ofrecer nuevas funcionalidades. En concreto, se ha establecido una infraestructura para que estas máquinas se comuniquen entre sí mediante mensajes, que son los mensajes que utilizan los microservicios internos de cada máquina. Esta infraestructura es el entorno de coreografía, y utilizando esta tecnología se obtiene un sistema ampliable ya que podemos añadir nuevas máquinas según las necesidades puntuales del sistema e incluso mover un microservicio de una máquina a otra. La tecnología que está detrás de la comunicación entre microservicios también permite conectar dispositivos entre sí. Así, una Raspberry pi puede utilizar los microservicios que están en otra máquina, en caso de requerirlos, como si estuvieran en la misma máquina física. El avance que supone añadir esta capa frente a desarrollos tradicionales es claramente visible puesto que se está ofreciendo una solución que sirve para modelar cualquier tipo de problema real y que posee muchas funciones ya implementadas.

Además de los avances que aporta la tecnología de coreografía, también se han realizado implementaciones que permiten que sensores de cualquier tipo se comuniquen con el sistema. Esto se hace a través de la creación de microservicios dedicados a cada una de las tecnologías utilizadas en la comunicación del mensaje. Por ejemplo, si tenemos un sensor que se comunica con la tecnología LoRa y necesitamos que nuestro sistema sea capaz de comunicarse

con ese sensor, se acoplará el hardware necesario para recibir datos de una red LoRa a una única Raspberry pi y se creará un microservicio que sepa utilizar ese hardware. Luego, los modelos establecidos para la transmisión de la información entre microservicios determinarán qué hacer con los datos recibidos.

En concreto hemos partido de un esquema básico formado por cinco microservicios representados por recuadros naranjas, como se observa en la imagen anterior. Estos microservicios tienen las funciones de:

- Comunicarse con las demás Raspberry pi para formar un sistema único: Microservicios *TCPListener* y *TCPConnector*. Un TCPConnector se conecta a un TCPListener de otra raspberry pi. El TCPListener local sirve para que otras Raspberry se conecten a la actual.

- Recibir datos de la red de sensores (WSN): Microservicio *WSNHandler*

- Gestión de los datos a través de una pequeña base de datos SQLite local: Microservicio *DatabaseService*

Cada uno de los microservicios nombrados se comunica con los demás microservicios para realizar una función. La comunicación se hace por medio de mensajes de coreografía y un modelo de comunicación. Los anteriores microservicios utilizan distintos modelos de comunicación según sus necesidades. Por ejemplo, los microservicios TCPListener y TCPConnector se conectan con sus homólogos en otra Raspberry pi formando un par TCPListener-Connector por donde se intercambian los mensajes de coreografía por encima de una capa TCP. Estos microservicios utilizan el protocolo REQ-RES para realizar esta función. El microservicio *DatabaseService* ofrece distintos patrones de comunicación, tanto por suscripción (patrón DATACENTRIC) como con reglas inteligentes (INTELIGENTE). El sistema base se puede ampliar. Podemos añadir nuevos dispositivos físicos con su conjunto propio de microservicios, o podemos añadir nuevos microservicios a los dispositivos existentes. También podemos trasladar los microservicios de un sitio a otro sin que surja ningún inconveniente en la programación.

Ya que partimos de un modelo de comunicación relajado, las comunicaciones entre microservicios no vienen dadas previamente y son los propios microservicios quienes interactúan con el sistema para registrarse o conocer nuevos microservicios según sea su interés por participar en el sistema. Este proceso no sería posible sin la existencia de un agente colaborador, el bróker. El bróker se encargará de realizar este trabajo; para ello mantiene el estado actual del sistema, en el que se incluyen las conexiones existentes entre microservicios y qué microservicios están ejecutándose. También es el bróker quien facilita la dirección del microservicio con el que nos deseemos comunicar, como se analizaba en las secciones previas, en el proceso de descubrimiento que ofrece este microservicio.

5.4.1. Relación entre mensajes y microservicios

En esta sección se toman en cuenta los tipos de mensajes actuales, los microservicios que forman el sistema y los tipos de microservicios que pueden participar. Se pretende relacionar todos los términos y dar sentido al funcionamiento global del sistema. Para ello, se divide la funcionalidad en distintas cuestiones que se resolverán a continuación.

A nivel lógico, nuestro sistema está formado por un conjunto de microservicios virtuales desarrollados con Java que se comunican entre sí. Para hacer posible esta comunicación, los microservicios utilizan mensajes de coreografía. Además, existen distintos tipos de microservicios, según la manera en la que realicen el envío o la recepción el mensaje. Por eso, es importante mencionar, que para que los microservicios se conozcan entre sí, existe un único microservicio con una funcionalidad adicional, que es el bróker. Este microservicio, que tiene una dirección conocida por todos los otros microservicios, ofrece un protocolo para que estos microservicios se comuniquen entre sí. Este protocolo define un proceso de publicación y descubrimiento, en el que los microservicios publicadores se registran en el bróker y los suscriptores preguntan al bróker y este les asigna las direcciones de los publicadores correspondientes. Para poder entender mejor cómo se definen y trabajan internamente los microservicios en este entorno, sugerimos las siguientes preguntas:

¿Cómo se define cada tipo de microservicio?

Nuestro sistema contempla tres tipos de microservicios según el protocolo interno que utilicen para comunicarse. En nuestro caso se han definido tres protocolos o modelos de comunicación que determinan cuándo y cómo se debe transmitir la información. Estos modelos son el REQ-RES, DATACENTRIC, y el modelo INTELIGENTE. No son excluyentes, lo que significa que un microservicio puede utilizar más de un modelo o patrón en sus comunicaciones. Existen dos roles para todos los modelos, que son el rol de proveedor y el de consumidor, según deseen transmitir o recibir información. Esto es importante ya que el registro de microservicios en el bróker sólo registrará al microservicio proveedor, y en el proceso de descubrimiento será el consumidor quien le solicite al bróker la dirección del proveedor. Analicemos el primero de estos procesos para entender mejor su funcionamiento, pues en todos los casos este proceso está compuesto por el envío de un único mensaje HELLO al bróker:

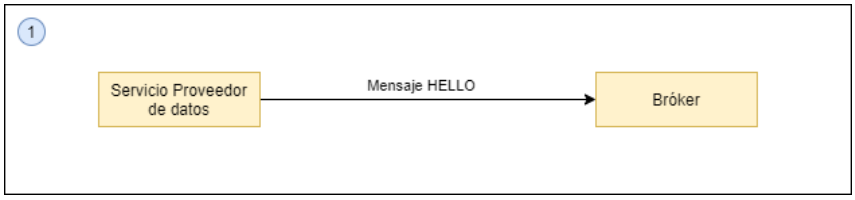

Figura 11. Mensaje HELLO

El bróker, que atiende los mensajes HELLO, realiza un análisis interno para extraer de ese mensaje toda la información que necesita para realizar el proceso de descubrimiento. Este proceso se consigue gracias a Regexp, que es el motor que permite analizar la estructura de los mensajes para determinar el tipo de microservicio que está registrándose.

En el anexo 1 se muestran algunos ejemplos de mensajes de todos los tipos implementados.

¿Cómo funcionan las máscaras?

Las máscaras son el mecanismo que utilizan los microservicios para comunicarse entre sí de manera “indirecta”. También permiten que se establezcan canales de comunicación sobre los que los microservicios pueden registrarse y comunicar datos. Una máscara es un identificador representado por una cadena de caracteres. Usualmente no se utilizan caracteres numéricos, sólo alfabéticos, debido a que son más descriptivos que los primeros.

Los mensajes de coreografía se deben etiquetar con una máscara. Además, todos los microservicios poseen una lista de máscaras. Aquellos microservicios en los que la máscara del mensaje concuerde con alguna de su lista de recepción recibirán el mensaje anterior. Por ejemplo, un microservicio que posea las máscaras “humedad” y “riego” recibirá los mensajes etiquetados con alguna de esos identificadores. A continuación, se muestra un ejemplo de un canal gracias a utilizar la etiqueta “Humidity”:

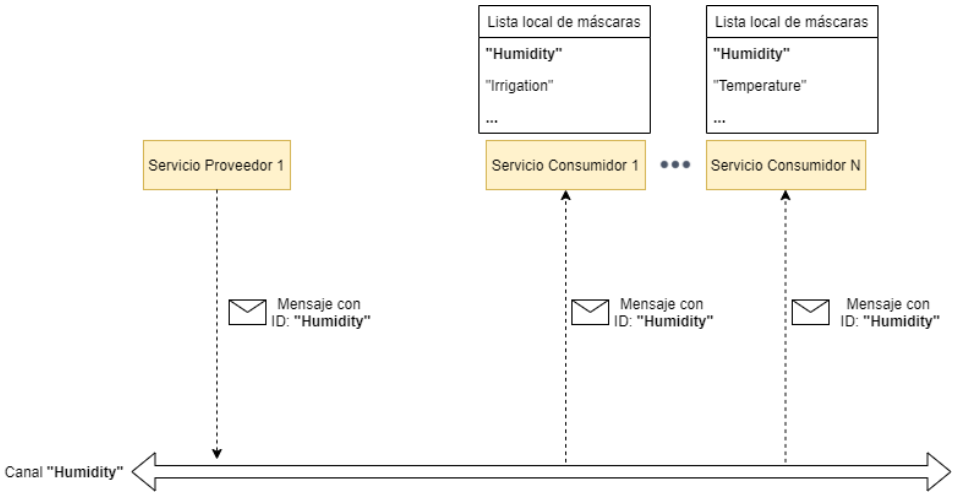

Figura 12. Ejemplo de uso de máscara

En el ejemplo anterior se establece un canal de comunicación de datos de humedad. Especificamos que todos los microservicios que publiquen datos de humedad deben utilizar el identificador "humedad", mientras que todos los microservicios que deseen algún dato de humedad añadirán a su lista interna de máscaras de recepción la nueva máscara "Humidity". De esta forma obtenemos un canal de publicación-suscripción sobre el que enviar o recibir datos. Simplemente todos los microservicios que conozcan el identificador 'Humidity' serán capaces de enviar y recibir mensajes en ese canal.

Además del ejemplo anterior, también se pueden establecer patrones de máscaras relativos, y para ello existe el carácter '.'. Este carácter divide la máscara en secciones, y permite enviar un mensaje a dos microservicios con máscaras distintas. Por ejemplo, dos microservicios tienen las máscaras "Communication.humidity" y "Communication.temperature" respectivamente. Un mensaje etiquetado con la máscara "Communication.*" será recibido por los dos microservicios anteriores, mientras que un mensaje etiquetado con "Communication.humidity" será recibido solo por uno de ellos.

¿Cómo entiende el bróker los mensajes? (Regexp)

El bróker es un microservicio inteligente que es capaz de entender los mensajes que recibe y saber si se trata de un proceso de registro/publicación o uno de suscripción. Se ayuda gracias a las palabras clave de cada tipo de mensaje. Posteriormente realiza un análisis del contenido en función del proceso al que pertenezca el mensaje. El proceso de publicación hace lo siguiente:

i. Discriminación de tipo de microservicios

Dependiendo del tipo de cadena (Lenguaje natural o estructurada) se realiza uno de los dos procesos:

a. Cadena en Lenguaje Natural

Primero se divide la expresión en subcadenas, cortando en las conjunciones, disyunciones, y cambio de frases. ('and', 'or', '.').

Split at	"AND", "OR", "."

Por cada subcadena resultante del proceso anterior se le asigna un tipo de microservicio dependiendo de la palabra clave que contenga (no distingue mayúsculas y minúsculas).

Tipo de microservicio	Palabra que debe aparecer
REQ-RES	“WHEN”, “REQ”,”REQ-RES”, “REQ”,”RES”,”RECV”
DATACENTRIC	“EVERY”,”CHANNEL”,”CH”,”C”
INTELLIGENT	“RULES”

b. Cadena estructurada

Las subcadenas vienen ya dadas en forma de array y no se necesita ningún proceso de división en subcadenas complejo, como en el proceso anterior.

Por cada subcadena, el contenido del atributo ‘P’ indica el tipo de microservicio.

Tipo de microservicio	Valor de atributo ‘P’
REQ-RES	“REQRES”
DATACENTRIC	“DATACENTRIC”
INTELLIGENT	“RULESERV”

ii. Extracción de atributos

Para cada una de las subcadenas anteriores existirá un proceso de extracción de atributos (o parámetros). En el caso de Leng. Nat.:

Atributo	Dominio
CHANNEL	Integer
FREQUENCY	String
RULESTYPE	TRESHOLD, AVERAGE, TENDENCY, MAXMIN
SENSORTYPE	String

En el caso de String Estructurado el valor de cada atributo es el valor del parámetro al que representa en la estructura:

Atributo	Parámetro en estructura
CHANNEL	“Ch” = Integer
FREQUENCY	“F” = String
RULESTYPE	“R” = String
SENSORTYPE	“M” = String

La gestión del bróker de HELLO-BYE:

El bróker implementa un protocolo interno de mensajes que tiene como objetivo reducir el tráfico en la red de coreografía y maximizar la eficiencia del sistema. Este protocolo está formado por los mensajes HELLO y BYE, y sirve para determinar la actividad de los suscriptores y publicadores. El mensaje HELLO indica que un nuevo proveedor de datos se ha

registrado/publicado en el sistema y por lo tanto cualquier microservicio que lo desee se puede comunicar con él. El bróker mantiene una lista de los microservicios consumidores conectados a cada microservicio proveedor. En el momento en el que un proveedor finalice la emisión de datos, el bróker debe avisar a todos los microservicios de la lista del proveedor con un mensaje BYE. De esta forma, los microservicios clientes saben cuándo el proveedor ha dejado de estar activo y pueden actuar según deban. Este mensaje BYE, aunque no aporta ninguna funcionalidad interesante a los patrones de comunicación REQ-RES y DATACENTRIC ya que el cliente tiene otros mecanismos para detectar la desconexión (ya sea porque sobrepasa del tiempo entre envíos del patrón DATACENTRIC o porque no se recibe un mensaje de un microservicio que atiende peticiones REQ-RES), es crítico para el patrón INTELIGENTE. En este patrón, los receptores de datos no pueden diferenciar si se encuentran en un estado de no-recepción de datos porque no se cumple la regla o porque el microservicio no está activo. Mediante el mensaje BYE obligamos a que los microservicios publicadores envíen un mensaje al bróker cuando finalicen su ejecución, y así el bróker puede avisar a los microservicios con un mensaje BYE, indicándoles la desconexión del microservicio proveedor. A continuación, se muestra un ejemplo del algoritmo de mensajes HELLO-BYE en el que interactúan un proveedor, un consumidor de datos y un bróker. El algoritmo se divide en dos partes; En primer lugar, la parte de creación de la lista y adición de nuevos suscriptores:

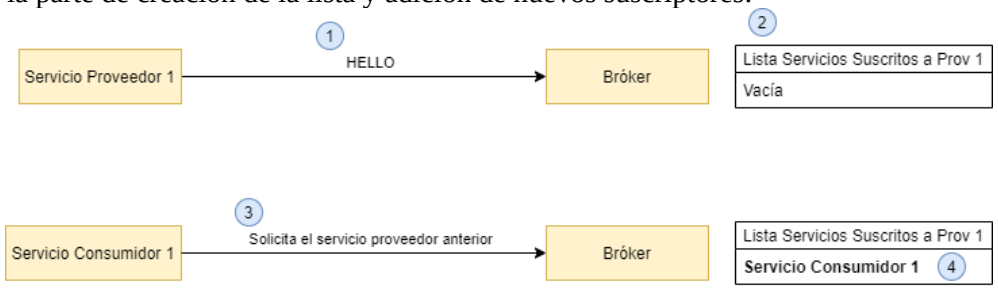

Figura 13. Ejemplo HELLO-BYE parte 1

A continuación, se explican los puntos descritos en orden:

1. Publicación en el bróker de un nuevo microservicio proveedor de datos (Proveedor 1)
2. El bróker crea una lista dinámica vacía con el nombre del proveedor anterior
3. Después de un tiempo, un microservicio solicita una conexión con el Proveedor 1 al bróker
4. El bróker añade el microservicio consumidor a la lista de microservicios suscritos al microservicio Proveedor 1

En segundo lugar, se muestra la parte de desconexión del proveedor y el aviso del bróker a los suscriptores:

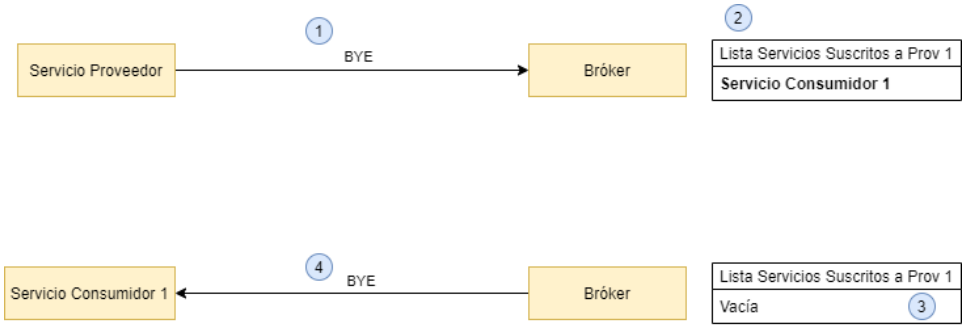

Figura 14. Ejemplo HELLO-BYE parte 2

Los pasos en este esquema siguen así:

1. El microservicio proveedor desea dejar de emitir y envía un mensaje BYE al bróker.

2. El bróker debe analizar la lista de microservicios suscritos, si hay alguno, entonces,
3. Por cada microservicio de la lista:
4. Envío un BYE al microservicio consumidor

De esta forma, el consumidor ya sabe que el proveedor ha dejado de emitir y puede actuar según convenga.

La gestión del bróker de mensajes de tipo PROBE:

El bróker atiende mensajes de tipo PROBE, que son los mensajes que envía el cliente para comenzar el proceso de suscripción a un microservicio. El bróker responderá al emisor del PROBE con un mensaje PROBE-MATCH, con las ocurrencias de los microservicios que encajen con la petición indicada en el PROBE.
El bróker mantiene una lista dinámica de todos los microservicios que proveen datos en el sistema. Esto es posible gracias al proceso de publicación que se encarga de añadir los nuevos microservicios registrados a esa lista. Cuando un cliente realiza un PROBE, está solicitando al Bróker un subconjunto de microservicios de esa lista. Ese subconjunto viene determinado por el tipo o modelo de datos que desee el microservicio que esté realizando la solicitud, la frecuencia de envío de los datos o incluso el tipo de sensor que realiza los envíos. A continuación, se presenta el esquema del anterior algoritmo:

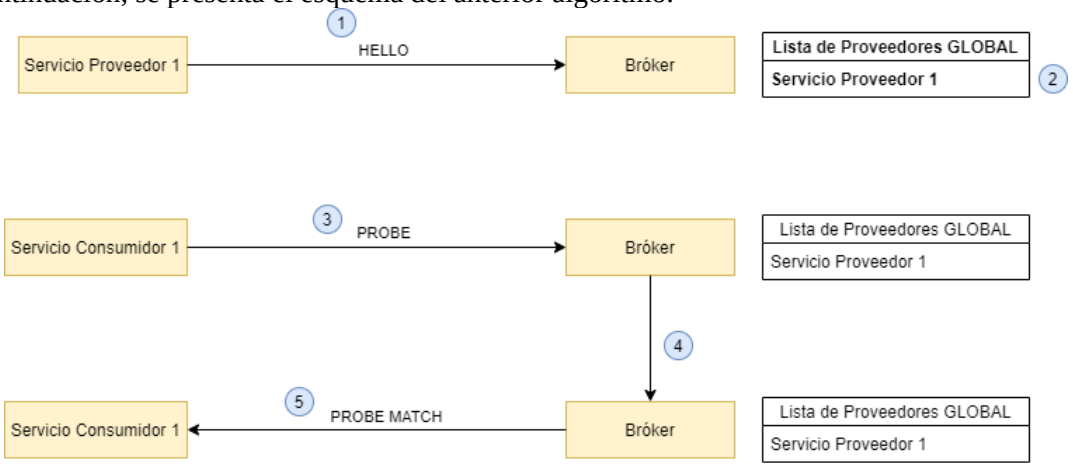

Figura 15. Ejemplo de mensajes PROBE

Los pasos son los siguientes:

1. Proveedor se publica en el bróker mediante un HELLO
2. El bróker actualiza la lista dinámica de proveedores y añade el nuevo microservicio a la lista
3. El microservicio consumidor solicita un conjunto de microservicios con el PROBE
4. El bróker analiza las necesidades del consumidor, y a través de un análisis REGEXP determina qué elemento o elementos de la lista dinámica debe devolver en el paso 5
5. Devuelve los elementos seleccionados en un mensaje de tipo PROBE-MATCH

A partir de este punto, el microservicio solicitante obtiene una lista de elementos que cuadran con su petición. Sólo tiene que escoger los que desee (como mínimo 1) en el proceso de RESPONSE.

La gestión del bróker de mensajes tipo RESOLVE:

El proceso de RESOLVE se inicia en el cliente, una vez que posee una lista de microservicios que atienden a su solicitud PROBE. Es trabajo del cliente determinar qué subconjunto de microservicios de esa lista le viene mejor. Ese subconjunto será enviado al bróker para que determine qué microservicio es el adecuado y se lo ofrecerá en la respuesta RESPONSE-MATCH. Se observa el algoritmo en el siguiente diagrama:

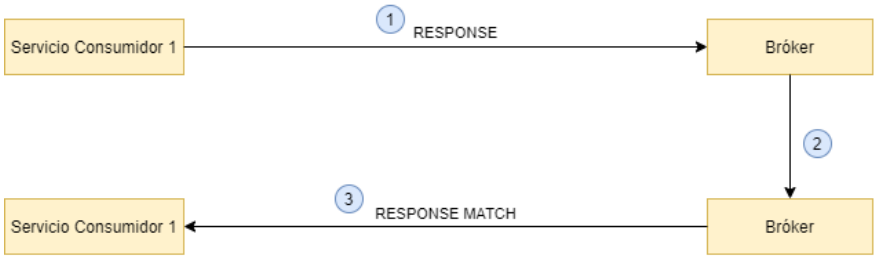

Figura 16. Ejemplo de mensaje RESPONSE-MATCH

Los pasos son los anteriormente descritos:

1. El consumidor devuelve un subconjunto de los microservicios recibidos en el proceso de PROBE.
2. El bróker analiza cuál de los microservicios es el más adecuado, teniendo en cuenta la carga de los microservicios y las necesidades del cliente.
3. El bróker responde al consumidor con un único microservicio que concuerda con la solicitud con la que inició el cliente el proceso de descubrimiento.

La gestión del aviso al publicador cuando no hay suscriptores:

El bróker mantiene una lista dinámica con el número de clientes que se comunican con cada proveedor de datos. Esto es posible gracias al proceso de publicación y descubrimiento. En el proceso de publicación, nuevos microservicios se dan de alta y se añaden a la lista de microservicios. En el proceso de descubrimiento, si un microservicio desea conectarse a otro y lo consigue mediante la secuencia de mensaje PROBE-RESPONSE, el bróker registrará en una lista individual por cada microservicio proveedor de datos, que un nuevo cliente se ha suscrito a él. De esta forma, no sólo podemos saber en tiempo real el número de microservicios de cada proveedor consultando la lista individual asociada a cada uno, sino que además podemos implementar funcionalidades que eviten la sobrecarga del sistema. En concreto, se ha implementado un mecanismo en el que el bróker detecta cuándo un proveedor no tiene ningún microservicio asociado a él y se le avisa con tal de que deje de emitir datos, puesto que no hay nadie que los reciba. A continuación, se muestra un diagrama de secuencia en el que se muestra claramente esta funcionalidad en la que intervienen tres agentes, el cliente, el bróker y el proveedor.

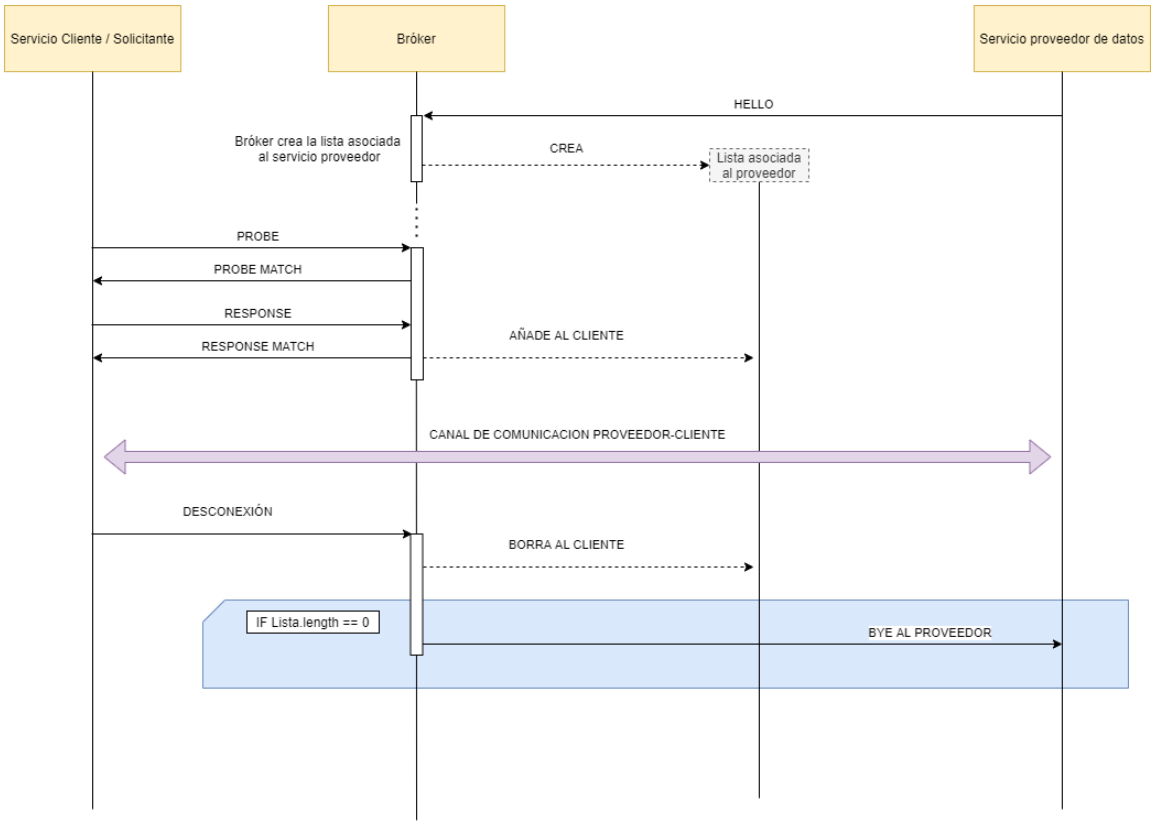

Figura 17. Diagrama de secuencia RESPOSE-MATCH

5.4.2. Base de Datos Firebase

En el esquema global del sistema existe una base de datos ubicada en un servidor externo. Esta base de datos tiene como único objetivo ser el *backend* de la aplicación móvil. Esto ha facilitado mucho el desarrollo puesto que, si hubiéramos tenido que realizar la conexión de la app directamente con el entorno de coreografía para obtener los datos sin contar con un servidor de bases de datos externo, hubiera supuesto mucho más trabajo ya que se habría tenido que implementar una API que debería estar por norma en un único microservicio, lo que dificulta las tareas de división del trabajo y concurrencia que necesita un entorno de coreografía. Enviar los datos al servidor de Firebase cuando se reciben es relativamente fácil; leer los datos de Firebase en la aplicación tampoco es complicado. Por tanto, el desarrollo en sí ha supuesto menos trabajo que lo que supondría implementar una API en un microservicio centralizado de coreografía.

Aun así, se ha debido de implementar un microservicio, pero no para que proporcione un acceso universal y centralizado a los datos, si no para enviar los datos recibidos en cada Raspberry pi a Firebase en tiempo real. Y, además, en lugar de haber un único microservicio en el sistema, existen tantos microservicios de envío de datos a Firebase como Raspberry pi formen el sistema. Esto se debe a que también se deben enviar los datos de las bases de datos locales de cada uno de los dispositivos a un sitio centralizado que posea toda la información.

Pero antes de hablar del esquema que sigue nuestra base de datos es necesario comentar ciertas características que hacen de Firebase la elección óptima para el desarrollo de nuestra solución.

En primer lugar, es necesario indicar que Firebase es una base de datos no relacional que ofrece muchas formas de trabajar con los datos. En concreto, se puede trabajar o con archivos JSON o con colecciones, y en nuestra solución hemos optado por trabajar con archivos JSON, debido a la gran cantidad de desarrollos funcionales con este lenguaje. Por otro lado, Firebase no sólo es un servidor de bases de datos, y esto es muy importante, ya que ofrece funciones adicionales a los servidores de datos tradicionales, como la autenticación, registro, etc.

La aplicación, que va dirigida a un gran número de usuarios, necesita algún mecanismo de autenticación y registro de nuevos usuarios. Esta funcionalidad de gestión de usuarios debería ser implementada, pero Firebase ya la posee, por lo que no ha sido necesario gastar recursos en el desarrollo de un agente autentificador en el servidor. Además, la brevedad de tiempo de acceso y el hecho de que se pueda trabajar con una gran cantidad de datos y usuarios gratuitamente facilita mucho más el desarrollo.

Por otro lado, si nos centramos en la estructura interna de la base de datos, observaremos el esquema de la base de datos que se especificó a partir del diseño requerido en la aplicación. Utilizaremos JSON para definir una estructura básica, necesaria para que la aplicación funcione. Esta estructura tiene tres nodos principales, “Gardens”, “Vegetables” y “Alerts”. El primer nodo contiene los datos de los sensores, el segundo nodo contiene los tipos de vegetales que la aplicación admite, y el tercero sirve para almacenar las alertas que puedan crear los usuarios sobre los datos de los sensores. En este esquema, cada Raspberry Pi se identifica con un único huerto y los datos que recibe esa Raspberry pi van a parar siempre dentro del mismo huerto.

Aunque la estructura básica de la base de datos se presentó en la sección de análisis de este documento, se hizo de manera muy superficial y únicamente se indicó la relación de jerarquías que sigue un modelo JSON. A continuación, podemos ver la estructura real de nuestro sistema desde los nodos base.

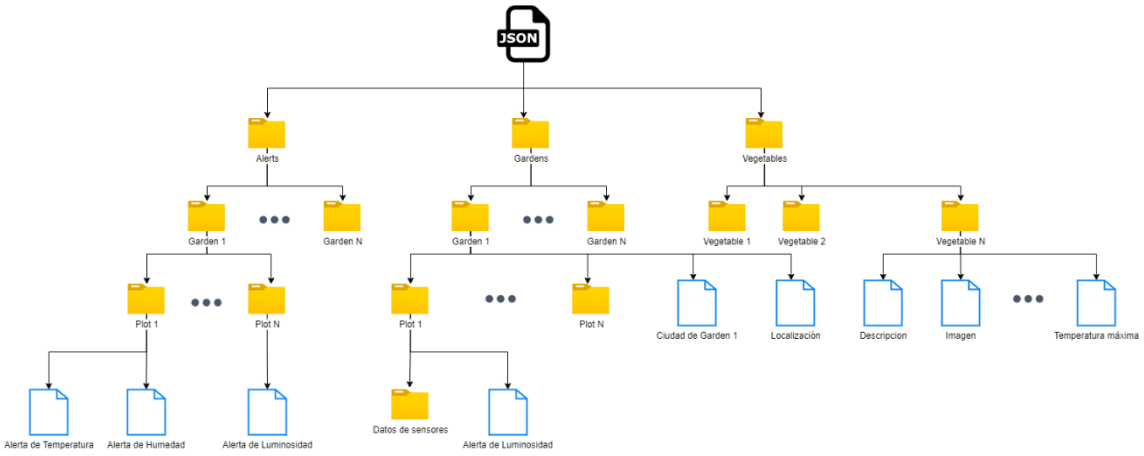

Figura 18 - Esquema global de la base de datos en Firebase

Se ha seguido un modelo que pretende simplificar los niveles de profundidad de la información a un límite de 4 niveles a costa de obtener un crecimiento lateral bastante elevado. Esto dificulta el desarrollo de la aplicación móvil puesto que las consultas deben ser algo más complejas, pero sirve para aportar mayor eficiencia al funcionamiento general de la aplicación ya que la frecuencia y retardo de acceso se ven reducidos enormemente con esta estructuración. Como se comentaba, el coste de añadir nuevos elementos al árbol no es elevado, pero requiere conocer la estructura general y distribuir los elementos entre las distintas secciones.

5.4.3. Seguridad en las comunicaciones

La seguridad en nuestro sistema se ha implementado en lugares distintos debido a que se trata de un sistema distribuido que pretende cumplir al máximo los estándares de seguridad. Dependiendo de donde nos situemos será útil un tipo de seguridad u otra. Por ejemplo, los requisitos de seguridad en la comunicación entre microservicios no son los mismos que los requisitos de seguridad de la base de datos. Tampoco tienen nada que ver con las necesidades de la seguridad de acceso a la aplicación y autenticación de los usuarios.

Como se ha podido comprobar, el análisis de la seguridad no está limitado a un único ámbito. Por ello vamos a distinguir entre los tipos de seguridad y los lugares donde se ha aplicado. A continuación, diferenciamos tres tipos de seguridad que han sido vitales para poder afirmar que nuestro sistema es 'seguro'.

5.4.3.1. Seguridad entre microservicios de coreografía

Todos los mensajes que envían y reciben los microservicios de coreografía que están en diferentes máquinas están cifrados, debido a que la comunicación entre máquinas se hace sobre una capa TCP, con lo que las comunicaciones pueden ser interceptadas Esto quiere decir que las comunicaciones entre todos los dispositivos de la capa EDGE permanecen encriptadas. Una persona que se sitúe entre dos microservicios que están en máquinas diferentes podría interceptar un mensaje utilizando técnicas de MITM. Gracias a la encriptación del mensaje, si fuese interceptado, no se podría interpretar.

El mecanismo que permite encriptar y desencriptar los mensajes está implementado en una librería que todas las máquinas de coreografía deben tener a su disposición. El cifrado que utiliza esta librería es simétrico, por lo que todos los microservicios deben compartir la misma contraseña. En concreto utiliza el cifrado RSA.

El uso de esta librería sólo se hace en los microservicios dedicados a interconectar las máquinas de la capa Edge, que son los microservicios TCPListener y TCPConnector. Estos microservicios reciben todos los mensajes de una máquina y los envían a otra máquina utilizando una conexión TCP, por lo que antes de enviar o recibir los mensajes deberán procesarlos. A continuación, se muestra una ilustración que presenta el proceso que realizan los microservicios TCPListener (recibe los datos) y TCPConnector (los envía) para transmitir un mensaje de un microservicio de una máquina a otro microservicio en otra máquina.

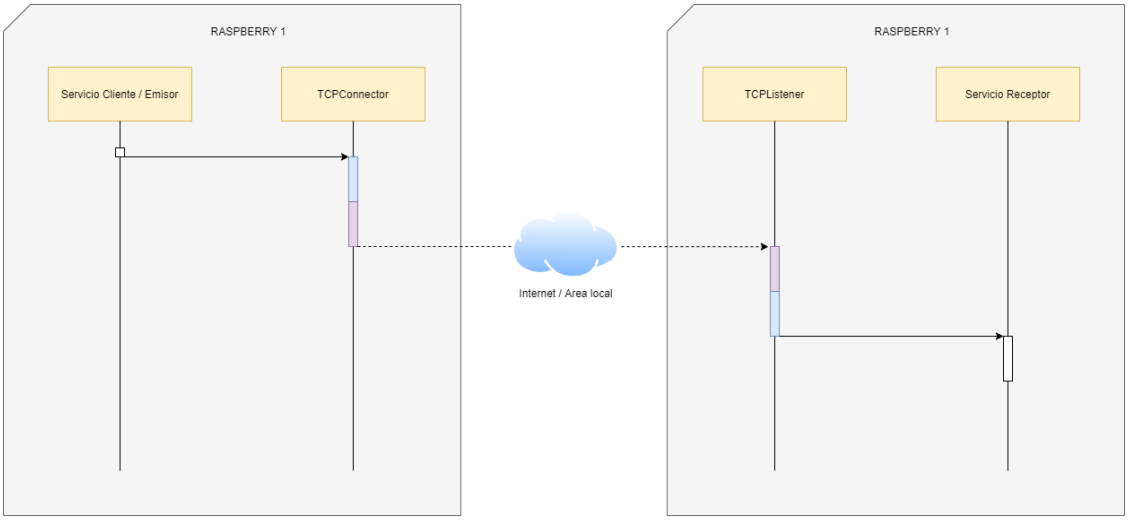

Figura 19 - Comunicación de mensajes de coreografía entre Raspberrry Pi

En el proceso anterior, el tcpConnector primero comprime el mensaje (sección azul) y luego lo encripta (sección morada). Por último, abre una conexión TCP y envía los datos comprimidos. El receptor, que es un microservicio TCPListener, recibe el mensaje, lo desencripta y lo descomprime. Luego analiza quién es el destinatario y entrega el mensaje.

El funcionamiento interno de ambos microservicios se puede ver a continuación.

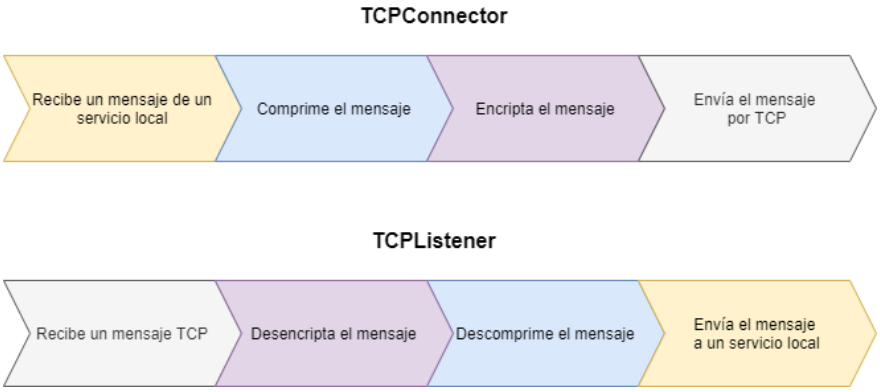

Figura 20 - Secuencia de trabajo de TCPListener y TCPConnector

En la imagen anterior, se distinguen las distintas fases que deben realizar ambos microservicios para comunicarse. La fase de compresión siempre irá antes que la de encriptación, por lo que la fase de descompresión deberá ir lógicamente después del proceso de desencriptado.

Por añadir, se podría extender el sistema actual para añadir una nueva característica de autenticación más segura que la actual, que se trata de una autenticación simétrica. Para ello sería necesario un microservicio que sea capaz de generar tokens de autenticación, y sería muy fácil de agregar a nuestro sistema. De esta forma, los microservicios primero se comunicarían con el microservicio de tokens para obtener su identificador de sesión y después comenzarían a trabajar en el entorno de coreografía ya autenticados.

5.4.3.2. Seguridad de acceso a la BD

La base de datos de Firebase ofrece diversas formas de acceder a ella, todas ellas con seguridad asociada. En concreto, hemos optado por la opción de autenticación mediante el SDK. Esta autenticación utiliza un archivo '.json' con nuestra clave pública y privada, que también contiene información adicional acerca del cliente que se va a conectar. Hemos elegido esta opción por ser la más popular, además de permitir un desarrollo más rápido. Así, para autenticar nuestra aplicación con la base de datos lo único que se necesita es situar el archivo JSON en una carpeta concreta y utilizar la librería de Firebase por defecto para Flutter. Para obtener el archivo json habrá que dirigirse al servidor de bases de datos, acceder con nuestra cuenta y crear una nueva aplicación asociada a nuestra base de datos. El proceso finalizará facilitándonos un archivo de configuración que se debe almacenar en el directorio del proyecto de la aplicación.

Seguridad de autenticación en la aplicación

Además de la seguridad de acceso a datos que se trata en el apartado anterior, se ha debido implementar una seguridad para gestionar el acceso de los usuarios de la aplicación (es importante no confundir el acceso a la aplicación con el acceso a la base de datos: en el primero se trata la posibilidad de acceder a la aplicación móvil completa y en el segundo se trata la seguridad de las consultas. Una vez se esté dentro de la aplicación, es decir, se haya autenticado como usuario, se deben analizar las consultas realizadas sobre la base de datos y determinar si el usuario tiene permisos suficientes. Esto ha permitido obtener un control y registro de todos los usuarios del sistema. Además, se pueden tener distintos tipos de usuarios con permisos diferentes. Por ejemplo, en nuestro caso contamos con dos perfiles distintos, el del alumno que no tiene permisos de escritura en la base de datos y el profesor, que sí puede modificar la base de datos. Aunque la funcionalidad de permitir o no modificar la base de datos está diseñada únicamente en la aplicación, la autenticación de los usuarios se ha hecho utilizando otra herramienta que ofrece Firebase. Esta herramienta se llama 'Authentication' y ofrece un módulo adicional en la base de datos que ofrece la autenticación de los usuarios existentes y un registro de nuevos usuarios. Así, simplemente habrá que implementar un par de llamadas a esta herramienta y el sistema de autenticación estará listo para funcionar. Además, el hecho de que este módulo se encuentre en la nube facilita que el administrador gestione los usuarios a través de cualquier navegador. Desde el servidor de bases de datos ahora se pueden crear nuevos usuarios, eliminarlos o modificar los ya existentes.

Siguiendo la línea de las necesidades de seguridad, que en nuestro sistema son muy básicas debido a que la información que se almacena en él no es realmente sustancial, hemos diseñado una única vía de registro por correo electrónico, pero fácilmente podemos añadir nuevos tipos de registro, como el registro por número de teléfono o incluso el registro mediante las cuentas de Google y Facebook que estén asociadas al dispositivo donde se ejecuta la aplicación. Aun así, la herramienta de autenticación de Firebase permite añadir nuevas vías de registro sin tener que programar una sola línea de código. También permite la verificación por número o llamada de teléfono y por correo electrónico.

5.4.4. Diseño de la aplicación

El objetivo de este apartado es dar a conocer en profundidad la aplicación móvil que se conecta nuestro sistema Edge. Esta aplicación, que se ha llamado *esGarden*, muestra los datos de una manera clara y ordenada. Hemos considerado de interés presentar las funciones más

importantes que se han implementado en nuestra aplicación, que es la única cara visible del proyecto a ojos de los usuarios.

Si nos centramos en los usuarios, la solución móvil está diseñada para un público joven en un entorno educacional. El objetivo de la aplicación es facilitar el aprendizaje de las tecnologías a las nuevas generaciones de estudiantes, además de acercarles al uso de la agricultura de una forma más elegante que las propuestas tradicionales. Es interesante que los jóvenes conozcan las tecnologías de los sistemas más actuales. Por eso se ha implementado esta aplicación, que es la encargada de ofrecer una visión global del sistema distribuido a los usuarios. De hecho, la aplicación es la única parte 'visual' del sistema, por lo que se ha realizado un análisis exhaustivo de posibles diseños a utilizar. Como la aplicación está diseñada para gente muy joven y sin conocimientos en informática, debe de ser minimalista y sencilla. Se han utilizado colores llamativos en las secciones donde se realizan las acciones y se ha optado por listas, iconos e indicadores visuales en lugar de información textual para facilitar una mejor comprensión. La aplicación, ofrece una interfaz compacta pero limpia.

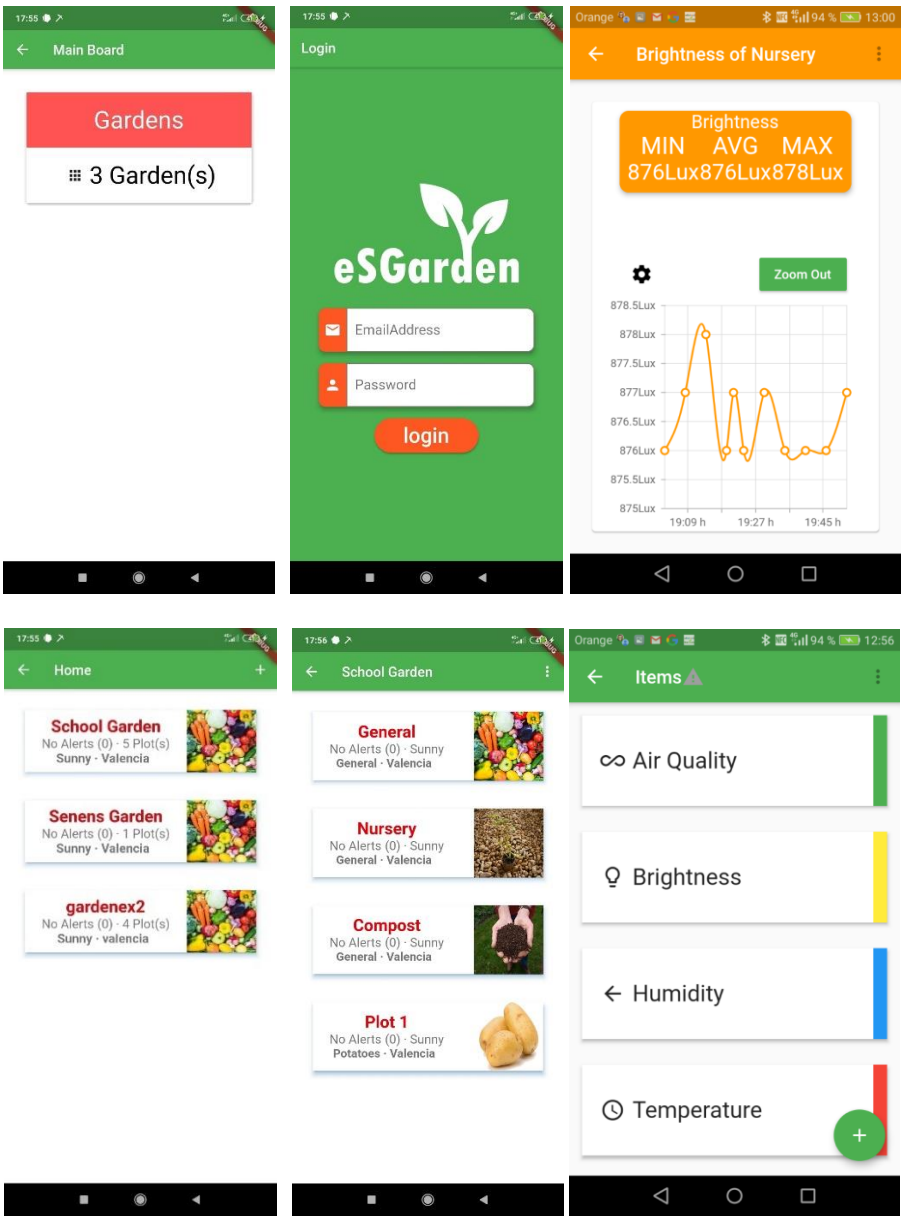

Figura 21 – Algunas pantallas de la app

5.4.4.1. Recorrido por la app

Nuestra aplicación necesita que los usuarios entiendan el modelo lógico que sigue. En él existen huertos, parcelas y sensores. Estos elementos deberán determinarse físicamente, ya sea sólo por el profesor, o con la ayuda de los alumnos. Es decir, habrá que dividir un terreno en parcelas e identificar cada una de las parcelas para evitar confusiones. Además, habrá que desplegar varios sensores por cada parcela y relacionar cada sensor con la parcela. Este proceso puede ser tedioso pero ayuda a que los alumnos comprendan mejor la división entre parcelas y de dónde proceden los datos.

La aplicación móvil distingue dos perfiles dentro de la aplicación. El perfil Alumno tiene limitadas ciertas capacidades en la aplicación que sí son accesibles al perfil Profesor. Para facilitar el registro de alumnos y profesores, se ha facilitado un código "secreto" que los profesores deben utilizar para registrarse como tal. El registro de un alumno se hace únicamente apretando el botón de login, sin necesidad de introducir usuario ni contraseña, ya que posiblemente no todos los estudiantes tengan un correo. Para resolver este problema, cuando un alumno se registra pulsando el botón, se guarda un token de identificación que sirve para autenticar a ese alumno durante toda la vida de la aplicación, a excepción de que desinstale y vuelva a instalar la app. En este último caso, bastará con volver a registrarse pulsando el botón "login". La app está disponible para poder ver todas las pantallas[10].

Después de registrarse, la siguiente ventana que se muestra es la ventana 'Home', que muestra la lista de huertos que están disponibles, con información adyacente como la ubicación, el número de alertas.

Si se aprieta sobre un huerto, podremos ver la lista de parcelas disponibles. Además, siempre habrá tres parcelas por defecto: Compost, General y Nursery: La pestaña General recoge datos que son generales al huerto, como por ejemplo la temperatura ambiente. Aunque este sensor deba de estar asociado a una parcela, hemos considerado que la temperatura y humedad ambiente en todo el huerto serán similares, por ello se muestran en esta pestaña. Por otro lado, las pestañas Nursery y Compost atienden a una petición de los colegios para conectar la aplicación con una máquina de compost y un pequeño invernadero de esquejes.

Para cada parcela podremos añadir o eliminar sensores en la aplicación. Esta funcionalidad se muestra en la ilustración inferior. Los sensores deben de haberse desplegado ya físicamente en la parcela. Este proceso se explica con detenimiento en secciones posteriores. Aun así, la forma de añadir sensores en la aplicación es muy sencilla. Simplemente habrá que pulsar '+' y se mostrará una lista de los sensores disponibles para ese huerto. Al pulsar sobre algún sensor, se añadirá a la vista de la parcela.

Cada sensor recoge y muestra la información de una forma diferente. En concreto, todos los datos recogidos tienen un instante de tiempo y un dato compuesto por uno o más valores. Si este dato sólo contiene un valor, como por ejemplo la luminosidad, se mostrará una gráfica simple con los datos por día.

En cambio, si el dato contiene varios valores, como es el caso de los datos de humedad de la tierra recogidos por sonda, que están compuestos por una tupla de tres valores de humedad a distintas alturas, se mostrará una gráfica compuesta con los datos superpuestos. Esto sirve de

[10] https://play.google.com/store/apps/details?id=myschool.esgarden&hl=en&gl=US

gran ayuda para determinar en qué momento estamos realizando una irrigación excesiva o si la humedad del subsuelo es muy elevada.

A parte de los sensores que recogen datos, también hay sensores que actúan sobre el huerto. En nuestro caso, sólo existe un sensor que cumple esta condición, el sensor de electroválvula. Si abrimos la ventana de la electroválvula de la parcela podremos establecer unos límites para regar autónomamente.

También se ha implementado una funcionalidad que permite crear alertas que notifican al usuario cuando se cumple una regla sobre algún valor recibido de los sensores. De esta forma, la aplicación nos avisará si alguna regla se cumple.

Además, los usuarios pueden ver el histórico completo de los datos en la base de datos para un tipo de sensor. La aplicación mostrará una gráfica con todos los datos existentes de este tipo.

Por último, para saber qué verdura se está plantando en cada parcela, se ha creado una entrada adicional en la lista de sensores de la parcela que se llama 'Vegetable'. Cuando se pulsa sobre esta entrada, se muestra la información del vegetal plantado de forma didáctica.

5.4.4.2. Casos de uso

Para completar la especificación de la aplicación, resulta de utilidad realizar un modelado de casos de uso que permita visualizar los roles y distinguir las diferentes funciones a las que tiene acceso cada usuario. Este diagrama sirve para determinar con claridad qué funciones son accesibles a cada rol.

Un diagrama de casos de uso focalizado en una aplicación está compuesto por distintos componentes. Estos son los actores, las unidades funcionales (o casos de uso) y las relaciones. Los actores representan a los usuarios que interactúan con la aplicación y los casos de uso hacen referencia a las funciones que cada usuario puede hacer. Para determinar qué funciones puede hacer un usuario utilizamos las relaciones. Además, es posible que un caso de uso sea utilizado por más de un actor.

A continuación, se muestra el diagrama de casos de uso de nuestra aplicación.

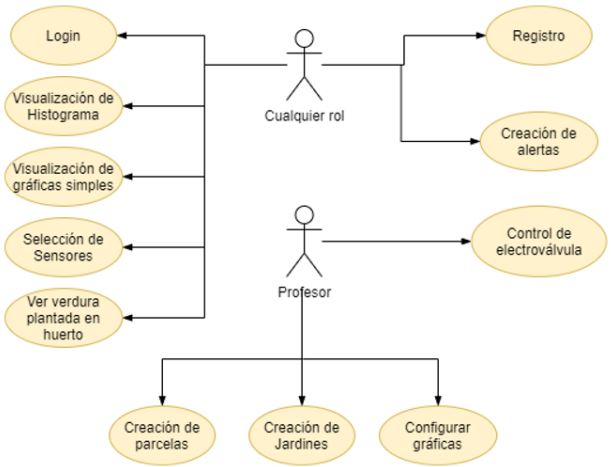

Figura 22. Diagrama de casos de uso de la aplicación

En él se observan los diez casos de uso y dos actores, que representan a los roles de Alumno y Profesor. Los casos de uso relacionados con el usuario Cualquier rol indican las funciones que tiene accesibles ambos roles. En nuestro caso, el usuario Profesor hereda todas las funciones de alumno y además se le añaden las operaciones de gestión, que permiten cambiar la configuración general de la aplicación, crear los huertos y parcelas que forman la aplicación y controlar un sistema de riego a través de la electroválvula. Además, ambos usuarios tienen la posibilidad de visualizar las gráficas en tiempo real de los datos del huerto y obtener una gráfica completa de todos los datos del huerto mediante la función de visualización del histograma. También ambos usuarios tienen la posibilidad de añadir y quitar sensores sobre cada parcela dentro de la aplicación, proceso que hay que hacer cada vez que se añada o elimine un sensor físicamente en el huerto.

5.5. Tecnologías utilizadas

Debido al carácter de nuestro proyecto, que está enfocado a la investigación, la cantidad de tecnologías utilizadas, entre todos los ámbitos a los que pertenece, es muy extensa. También hay que matizar que el uso de sistemas que son muy novedosos o que tienen poco tiempo de vida facilita la aparición de problemas, por lo que se convierte en un proceso muy tedioso y poco eficiente en general. Esto puede deberse a la poca información existente o a la falta de experiencia; aun así, es posible que funcionalidades que antes eran imposibles de adaptarse a nuestro proyecto ahora sí lo sean. Se propone la siguiente división según la capa donde se encuentre el aporte realizado.

5.5.1. Desarrollo en capa Edge

El desarrollo de la capa Edge ha significado la utilización de gran cantidad de sistemas novedosos pero muy poco maduros, por lo que se han debido realizar muchas pruebas para garantizar que todo funcione correctamente. La herramienta software que ha permitido crear el entorno Edge es el sistema de coreografía, desarrollado por el instituto SABIEN de la UPV. Esta herramienta se apoya sobre un hardware para poder funcionar de forma distribuida, es decir, ejecutarse de forma conjunta en varios sistemas. Así, obtenemos un sistema de microservicios distribuidos entre tantas máquinas como se deseen. El hardware en cuestión está compuesto por Raspberry pi, que están conectadas a la misma red local. Además, en la solución propuesta también se cuenta con un hardware adicional que se conecta a cada una de las Raspberry pi y que permite que reciban datos de una red local de sensores con LoRa.

5.5.2. Arduino y LoRa

Para crear la red de sensores que permite recoger los datos del mundo físico hemos utilizado varios Arduinos interconectados entre sí a través de la tecnología LoRa. LoRa es una tecnología de radiofrecuencia que permite que dos dispositivos se comuniquen entre sí a una gran distancia utilizando una cantidad ínfima de energía para realizar el envío de un mensaje. LoRa define una comunicación por radiofrecuencia a muy baja frecuencia, lo que significa que se necesita una cantidad insignificante de energía y supone un alcance muchísimo mayor, pero a costa de disminuir el tamaño de los mensajes a unos pocos bytes. Aunque los avances de utilizar esta tecnología para desplegar la red de sensores son claramente visibles, es importante mencionar que así se consigue una red con un mantenimiento insignificante, en la que los sensores no deben de estar próximos a una fuente de energía. Gracias al bajo consumo de esta tecnología podemos realizar envíos constantes con una pila AA durante varios años y situar sensores en lugares inaccesibles, pero hay que reconocer que el tamaño de los mensajes debe ser muy reducido y no exceder de un límite.

La arquitectura a más bajo nivel de nuestro sistema consiste en una red de sensores y conectados por LoRa a cada Raspberry pi. Los sensores se han implementado con Arduinos. Para adaptar los Arduinos a LoRa se ha utilizado el Shield que se muestra a continuación:

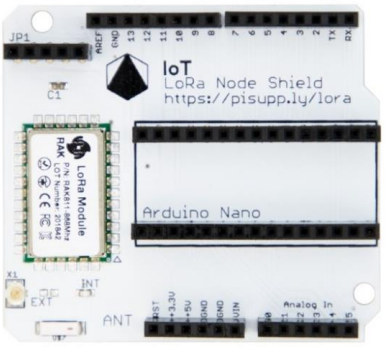

Figura 23 - Placa de comunicaciones LoRa de marca Pi Supply

Existe una gran cantidad de proyectos públicos que utilizan este 'sombrero' y que han servido como base para desarrollar el módulo de comunicaciones LoRa. Gracias al creciente número de proyectos desarrollados en este campo se han ido creando distintas librerías que sirven para que el Arduino se comunique de forma simple con el módulo LoRa.

Ahora bien, es importante mencionar que, aunque en nuestro proyecto estamos utilizando la tecnología LoRa, no se debe confundir con el protocolo de comunicaciones LoRaWan. Mientras que LoRa define la tecnología de comunicaciones entre dos puntos, la especificación LoRaWan define distintos elementos e interacciones entre ellos, tales como unas puertas de entrada (o Gateways) que deben ser públicas y conocidas por todos (pero no los datos que reciben, que son privados para cada usuario), unos sensores y un servidor centralizado. Las relaciones entre estos elementos vienen descritas en la especificación LoRaWan, y se tienen en cuenta aspectos tales como el tamaño de una unidad de mensaje o la seguridad entre las comunicaciones.

5.5.3. Microservicios de enlace

Previamente se han tratado varios de los microservicios que utiliza nuestro sistema y que no están físicamente en él. En concreto hemos hablado de un microservicio de bases de datos que se encarga de almacenar la información en un sitio externo al sistema, y que servirá para atender las peticiones de la aplicación móvil asociada. Pero además de este microservicio de almacenamiento de la información, existen una gran cantidad de microservicios que podrían aportar una funcionalidad extra en nuestro sistema. Gracias a haber diseñado la comunicación entre microservicios utilizando el estándar de WS-Discovery, cualquier microservicio externo que siga ese estándar podrá acoplarse a nuestro sistema. Además de ser compatible con microservicios que utilicen el estándar, sería relativamente fácil añadir nuevos microservicios en la nube implementando una conexión desde el despliegue de un nuevo microservicio interno. En resumen, es posible conectar cualquier microservicio con una API conocida a nuestro sistema, pero será nuestro trabajo desarrollar la interacción con ese microservicio en un microservicio de coreografía.

5.5.4. Flutter y Android Studio

Una herramienta de gran utilidad es la aplicación móvil, que permite a los usuarios obtener en tiempo real una visión global del estado del sistema. De esta forma, es más fácil detectar las diversas situaciones que se pueden dar en un huerto, e incluso se pueden programar alertas que

avisen al usuario cuando una condición se cumpla. Así, se puede avisar al usuario cuando haga falta realizar un riego, por ejemplo. Pero la idea original de la aplicación no se centra únicamente en ofrecer una vista del sistema real, sino que además se adapta a un modelo educativo focalizado a los más jóvenes. En concreto, el objetivo está en que los estudiantes sean conscientes y responsables con el medio ambiente a través de una aplicación atractiva que les permita conocer nuevas tecnologías sin realizar un esfuerzo muy elevado. Para ello se han determinado patrones visuales e identificadores que sean sencillos y fáciles de utilizar. Para realizar tales patrones visuales ha sido de gran utilidad la herramienta Flutter, que permite realizar aplicaciones con un gran valor visual en muy poco tiempo. Flutter, que es una herramienta de Google, se trata de un conjunto de librerías, o framework que facilita la programación en un lenguaje específico,que en este caso es Dart, que es un lenguaje optimizado para front-end. En su conjunto se obtiene una herramienta muy potente y que permite, mediante el uso de dart, desplegar aplicaciones con un front-end de gran calidad y con gran capacidad para el back-end. También, al ser desarrollada por Google, es ampliamente compatible con sus herramientas como Firebase o Google AdMob. El hecho de utilizar esta tecnología facilitará su acoplamiento en el futuro con otras tecnologías.

El código se ofrece en abierto para utilidad de la comunidad[11].

5.5.5. Play Store

La aplicación desarrollada en Flutter se ha subido a la tienda de aplicaciones de Play Store. Para ello se ha registrado una cuenta de desarrollador que ha supuesto una inversión de 25$. Para realizar el registro, Google ofrece un formulario en el que se puede asociar una cuenta de Gmail ya existente.

Play Store ofrece un panel de control que llama ‘Console’ para gestionar todo el ciclo de vida de una aplicación. Este panel permite desplegar la aplicación en una versión no final para que el sistema pueda probarse con usuarios reales y así realizar un mejor análisis de errores. La consola reporta los errores que puedan haberse ocasionado en los usuarios de prueba y facilita el análisis y localización de errores de programación o de despliegue. Además, la consola también nos permite visualizar el número de descargas y la localización de estas, con lo que se pueden realizar varios análisis, incluidos análisis indicados para el posicionamiento web (SEO en ingés). También es posible realizar actualizaciones sobre una aplicación, de tal modo que cuando se suba la nueva versión se avise a los usuarios para que la descarguen, e incluso se puede restringir el uso de la app a la última versión. En resumen, el hecho de utilizar la consola de Google facilita y reduce enormemente el proceso de publicación de las apps, además de ofrecer herramientas adicionales que exceden de la funcionalidad de subir una app al mercado o actualizarla.

5.6. Interacciones

A continuación, se describe el proceso de implantación dividido en dos secciones, la interacción Raspberry pi - Firebase y la interacción entre Firebase y la app.

[11] https://github.com/senenpalanca/esgarden

5.6.1. Interacción Raspberry pi-Firebase

La interacción entre las Raspberry pi y Firebase se lleva a cabo con un único objetivo: Llevar los datos nuestro sistema a una base de datos en la nube para que sea posible leerlos desde una aplicación Android. Pero ¿y por qué no conectar directamente la aplicación Android a una API de nuestro sistema?

Previamente hemos comentado la posibilidad de añadir esta capa a nuestro sistema, pero entonces deberíamos implementar un *backend* funcional que estuviese alojado en nuestro sistema distribuido y conectado a una base de datos local de grandes dimensiones, tarea casi imposible debido a la levedad de los dispositivos que lo forman, las Raspberry pi. La estimación de tiempo necesario para realizar esta opción es muy elevada y se trata de un desarrollo complejo, por lo que descartamos esta opción y optamos por conectarnos con un *backend* externo y únicamente implementar una interfaz de transmisión de datos a ese *backend*. Esta interfaz debe estar implementada en todos los componentes de nuestro sistema ya que nuestro almacenamiento está distribuido en bases de datos ubicadas físicamente en cada dispositivo. La interfaz se ha implementado por medio de un microservicio se comunica con Firebase siguiendo sus premisas de seguridad y frecuencia de envío. El microservicio ubicado en cada Raspberry pi, realiza una petición a la base de datos local, recupera los datos y los envía al *backend*.

5.6.2. Interacción Firebase-app

La interacción entre Firebase y la aplicación se realiza mediante una librería que se llama *firebase-core*. Esta librería ofrece todos los mecanismos necesarios para trabajar con bases de datos no relacionales que almacenan su información en formato JSON. En concreto Firebase te ofrece un gran árbol con un elemento raíz del que cuelgan todos los demás elementos. Cualquier dato de la base de datos será un elemento hijo del elemento Raíz. El proceso por defecto de nuestra aplicación es recoger cada uno de esos elementos y mapearlo a un objeto con tal de poder trabajar con él. Los elementos se organizan en tres grandes ramas que distinguen los tipos de datos almacenados en cada una. Así, cuando la aplicación necesite mostrar datos de una gráfica, obtendrá la rama correspondiente a los datos y no obtendrá información adicional que no necesita y que congestiona el tráfico en la red.

5.7. Instalación y puesta en marcha

Una de las características más importantes de nuestra solución es la flexibilidad. No proponemos un proyecto con una implementación específica, sino que diseñamos un sistema que se pueda adaptar a las distintas condiciones que puedan darse en un sistema distribuido enfocado a IoT, dejando al usuario la opción de realizar el despliegue como mejor le convenga y, de esta manera, pueda aprovechar al máximo las características de nuestra solución.

El reto ha sido convertir un proyecto que conlleva un profundo estudio y desarrollo en un despliegue que no requiera -por parte del usuario- de conocimientos avanzados. Más bien, está preparado para que cualquier persona con unas mínimas nociones tecnológicas tenga la

capacidad de desplegar el sistema y utilizarlo sin mayor problema. Esto es un requisito básico en nuestro sistema porque va destinado a un grupo heterogéneo de usuarios, los docentes de educación secundaria, cuya formación tecnológica e informática es muy diversa. Por eso, se les facilitará un breve manual con pasos guiados, que explicaremos en el anexo 2.

Dado que nuestro sistema es muy flexible y se pueden realizar distintas aproximaciones con él, proponemos una aproximación válida en la que participarían todos los componentes de nuestro sistema: los sensores, el entorno de coreografía, el servidor Firebase y la aplicación móvil. En esta aproximación se utilizan 2 Raspberry pi que contienen el sistema de coreografía.

A continuación, se presenta el esquema del sistema propuesto:

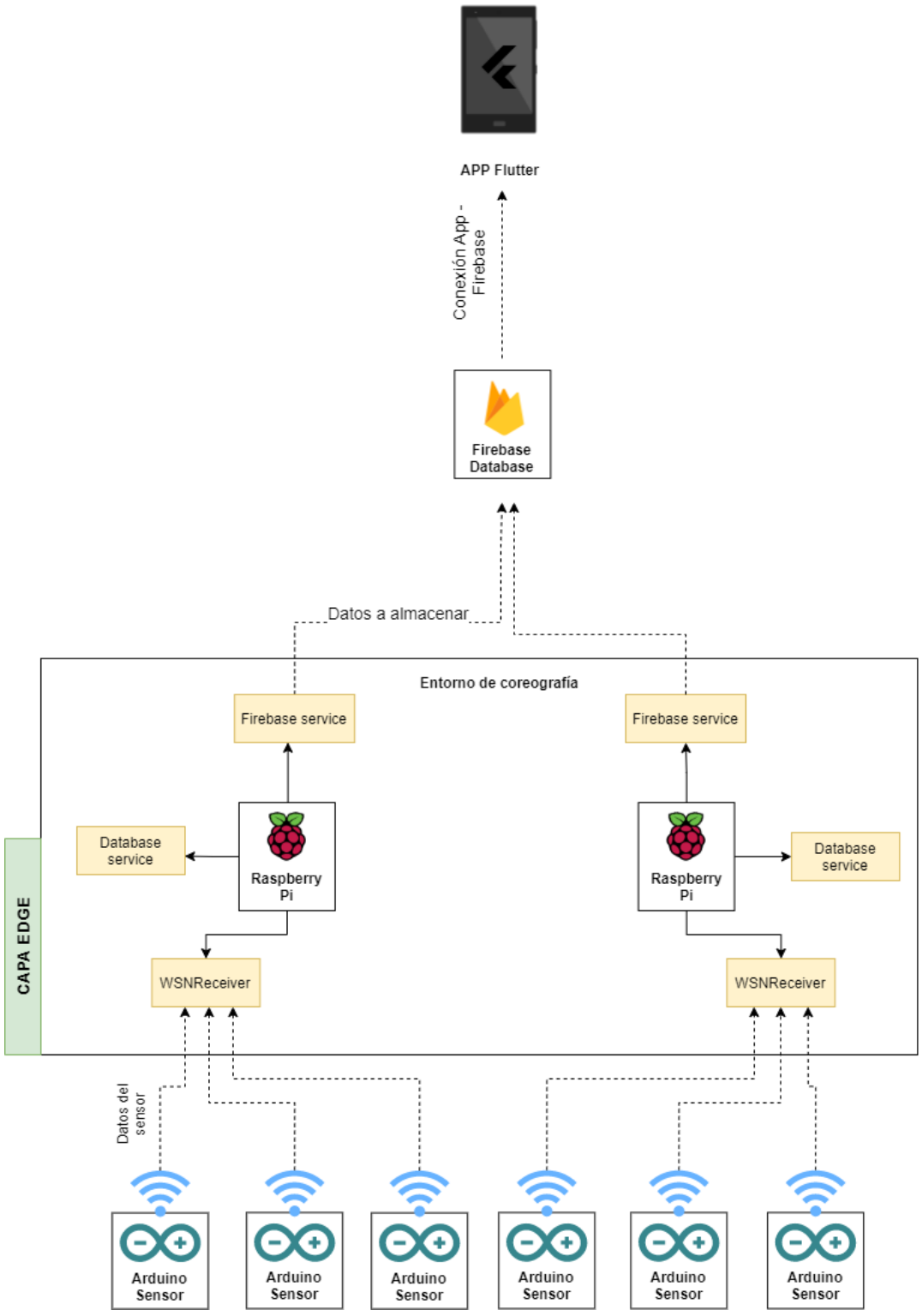

Figura 24 – Esquema del sistema prototipo

Como se observa, este esquema parte del esquema presentado en la solución propuesta y especifica varios microservicios en cada Raspberry Pi. Para facilitar el entendimiento se ha optado por un sistema básico en el que únicamente existen tres microservicios por Raspberry Pi. Estos microservicios tienen la función de recoger, almacenar localmente y enviar los datos al servidor de almacenamiento, que en nuestro caso es Firebase.

6. Implantación

La implantación de un sistema software hace referencia al proceso con el que se pone en marcha el sistema. Este proceso incluye las tareas necesarias para que el proyecto funcione correctamente. Nosotros consideramos que las tareas más importantes para la implantación son el análisis de funcionalidades del sistema y su despliegue. Por ello, es de gran ayuda analizar aspectos de la implementación de nuestra solución para obtener resultados de su funcionamiento y extraer conclusiones.

En nuestro caso, hemos diseñado tres análisis para garantizar que la solución aportada puede funcionar dentro de unos límites aceptables. Estos análisis se focalizan en las comunicaciones entre microservicios, la eficiencia de los patrones de datos y el análisis de expresiones simples. Además, se ha añadido un cuarto análisis dirigido a la aplicación, puesto que también hay que verificar que funciona dentro de un marco de seguridad y fiabilidad.

A continuación, se presenta el primero de los estudios que está relacionado con el análisis de expresiones por medio de Regexp, un mecanismo para el análisis de expresiones regulares que hemos implementado para extraer información de los mensajes de Publicación. En este estudio se observará el tiempo que tarda el sistema en analizar expresiones y atender las peticiones de los microservicios que utilicen expresiones regulares para comunicarse.

6.1. Análisis de latencias de Regexp

Si nos situamos frente a la comunicación entre el bróker y los microservicios, observaremos que estos últimos se comunican con el bróker para realizar dos tareas diferentes: o publicarse o suscribirse a otro microservicio del sistema. Como se comentaba previamente, para ambos procesos existe la posibilidad de realizar la petición en lenguaje natural. Aun así, también es posible hacerlo de forma estructurada siguiendo la especificación SOA para Servicios Web descrita en W3C (W3C). Para ello es necesario utilizar la notación definida en la tabla 1, que es la notación implementada en el proyecto y que significa lo siguiente:

serv = microservicio,
p = patrón,
f = frecuencia,
ch = canal,
r = regla,
m = modelo.

Tabla 1. Anotación semántica utilizada en los mensajes estructurados

Anotación	Descripción	Dominio	Valor
p	Patrón: solicitud-respuesta, centrado en datos y servidor de reglas	String	REQRES, DATACENTRIC, RULESERV
F	Tiempo: milisegundos, segundos, minutos, hora	String	ms, s, m, h
ch	ID de datos (sólo mensajes HOLA)	Entero	[00 ... 99]
r	Reglas de umbral, promedio, tendencia, maxmin	String	thrs, avg, tnd, max, min
m	Modelo o marca de sensor	String	ID del fabricante

A continuación, hemos realizado una comparativa entre las dos configuraciones de mensajes disponibles focalizando el estudio en el algoritmo que interpreta y extrae la información estos mensajes.

Por ejemplo, utilizando la tabla anterior, un microservicio de reglas (o INTELIGENTE) que puede proporcionar datos de humedad de acuerdo con un cierto umbral y que envía datos cada 10 minutos (sólo si se cumple la regla), redactará un mensaje de tipo HELLO para publicarse. A este mensaje se le llama el descriptor del microservicio, y el cuerpo del mensaje tiene la siguiente estructura:

<d: Tipos> i: humedad </ d: Tipos>
<d: Scopes> serv: /// p = RULESERV, f = 10m, r = thrs </ d: Scopes>

Es importante mencionar que un proveedor puede ofrecer más de un patrón de entrega y que un consumidor puede suscribirse a más de un patrón. Para ello, se define la posibilidad de enlazar en el mismo mensaje varias descripciones del microservicio, ya sea en formato estructurado o en lenguaje natural. También hay que recalcar que es imposible que un microservicio que no conozca la estructura común se comunique con el sistema, lo que supone un problema. Por este motivo se ha implementado la capacidad de atender peticiones de lenguaje natural mediante el análisis expresiones regulares. De esta manera podremos integrar nuevos microservicios con un conocimiento mínimo de la semántica utilizada en el sistema.

Sirve como ejemplo de un mensaje estructurado, el mensaje más simple que podemos hacer: un mensaje de tipo HELLO, que publica al microservicio como un microservicio que ofrece los datos mediante el patrón de REQ-RES. Además, el publicador tiene la opción de indicar si entrega únicamente el último dato o permite que se le hagan peticiones que devuelvan un conjunto de datos entre dos fechas o marcas de tiempo.

```
1.  XFIPAMSG_payload {
2.   <d: Hello>
3.   <a: EndpointReference>
4.    <a: Address> Service.address </ a: Address>
5.   </ a: EndpointReference>
6.   <d: Tipos> i: humedad </ d: Tipos>
7.   <d: Scopes> </ d: Scopes>
8.   <d: MetadataVersion> 0000 </ d: MetadataVersion>
9.   </ d: Hello>
10. }
```

Adicionales al anterior patrón existen otros patrones, como el DATACENTRIC o el de servidor de reglas (INTELIGENTE), que deben incluir aún más información en el cuerpo del mensaje. Además, un proveedor publicado fuera de la red local en la que se encuentra el sistema desplegado incluye parámetros XAddrs complementarios que sirven para comunicarse vía TCP en caso de que no existiese el entorno de coreografía. Como se observa en el ejemplo siguiente, este microservicio se corresponde con un microservicio de tipo INTELIGENTE, debido a que tiene dos parámetros indispensables, el porcentaje de la regla y la palabra ‘threshold’ que determina el tipo de regla a aplicar:

```
1. XFIPAMSG_payload {
2.  <d: Hello>
3.  <a: EndpointReference>
4.   <a: Address> Service.address </ a: Address>
5.  </ a: EndpointReference>
6.  <d: Tipos> i: humedad </ d: Tipos>
7.  <d: Scopes>
```

8. % de regla threshold
9. </ d: Scopes>
10. [<d: XAddrs> xs: localhost: 9020 xs: 192.168.1.215: 9031 </ d: XAddrs>]
11. <d: MetadataVersion> 0001 </ d: MetadataVersion>
12. </ d: Hello>
13. }

Estos ejemplos se corresponden con mensajes estructurados, y tienen la desventaja de que, si el microservicio desea expresar más de un patrón de datos, deberá anidar las peticiones en el campo *scope*. Esto hace que los mensajes sean mucho más extensos en comparación con los mensajes generados con cadenas en lenguaje natural, ya que en ese caso no es necesario anidar expresiones sino utilizar conjunciones gramaticales. Tomemos como ejemplo el siguiente mensaje que proviene de un tipo PROBE y que establece cuatro descripciones del microservicio que se necesita dentro del campo 'scope':

<d:scope>
Microservicio de solicitud y respuesta O microservicio datos cada 30 m desde el sensor STM32 O acepto reglas de umbral O acepto reglas de promedio
</ d:scope>.

El mensaje equivalente en nuestro código se muestra en la Figura 25:

```
public void Start()
{
    XFIPAMSG msgC = new XFIPAMSG(
       this.getIdentificadorAccion(),
       "Communication.Broker",
       XFIPAMSGTipoProtocolo.eventmsg
       );

    Dictionary<string, object> parametros = new Dictionary<string, object>();
    parametros.Add("Types", "i:humidity");
    parametros.Add("Scopes", "Req-res service OR data every 30m from STM32 "+
                             "OR rules of threshold OR rules of avg");
    msgC.Contenido = new SOAPMSG("Hello", parametros);

    this.OnPropagateCommand?.Invoke(this, msgC);
}
```

Figura 25. Ejemplo de código creado con una expresión regular en el campo scope: ejemplo de C#.

Por último, pasemos al análisis de latencias. Para ello debemos tener en cuenta el tamaño del mensaje. El mensaje anterior se corresponde con una única expresión en lenguaje natural que comprende las cuatro solicitudes es de 209 bytes. El tamaño del mismo mensaje, pero utilizando la configuración estructurada, forma un mensaje de 494 bytes. Esta diferencia, significativa en descripciones complejas, implica una mejora frente al uso de mensajes estructurados.

Por otro lado, la Tabla 2 compara varios mensajes HELLO donde:

- RE es el tiempo que tarda el bróker en resolver un mensaje escrito en lenguaje natural.
- S es el tiempo que tarda el broker en resolver un mensaje con la estructura presentada en la tabla 1.

La tabla realiza una comparación entre expresiones regulares y su composición equivalente con estructuras.

De manera similar se analizan los mensajes PROBE en el bróker. En la tabla 3 comparamos los retrasos del bróker con diferentes mensajes de clientes, según que el alcance sea una expresión regular o utilice cadenas estructuradas.

Tabla 2. Comparación de latencia RE y S: mensajes HELLO.

Alcance RE	Alcances S	RE	S
Enviar datos a pedido	p = REQ-RESP	24 ms	7 ms
Tipo de solicitud frecuencia de microservicio 5 m	p = REQ-RESP + f = 5 m	19 ms	8 ms
Req – microservicio de resp	p = REQ-RESP	24 ms	9 ms
Hola, soy un microservicio req	p = REQ-RESP	21 ms	10 ms
Enviar datos cada 30 m desde DHT11 en el canal 00	p = DATACENTRIC + m = DHT11 + f = 30 m + ch = 00	50 ms	27 ms
Enviar cada 1 h desde SGP40 en el canal 1	p = DATACENTRIC + m = SGP40 + f = 1 h + ch = 01	84 ms	32 ms
Frecuencia de datos 30 m en canal 02	p = DATACENTRIC + f = 30 m + ch = 02	64 ms	19 ms
Aceptar reglas de umbral	p = RULESERV + r = thrs	60 ms	17 ms
Acepta las reglas de la media	p = RULESERV + r = prom	62 ms	21 ms
Solicitar microservicio de resp y aceptar reglas de tendencia	String_1: p = REQ-RESP String_2: p = RULESERV + r = tnd	113 ms	78 ms
Reglas promedio, datos cada 30 m en cap 04	String_1: p = DATACENTRIC + f = 30 m + ch = 04 String_2: p = RULESERV + r = avg	135 ms	89 ms

ch: canal; thrs: umbral.

Tabla 3. Comparación de latencia RE y S: mensajes PROBE.

1	Mensaje PROBE estructurado	Respuesta PROBE-MATCH (coherente con campo type)	RE	S
-	-	Microservicio REQ-RESP	4 ms	3 ms
Solicitar cada 1 h	p = DATACENTRIC + f = 1 hora	2 microservicios DATACENTRIC	17 ms	9 ms
Datos cada 1 h desde SGP40	p = DATACENTRIC + f = 1 hora + m = SGP40	1 microservicio DATACENTRIC	20 ms	10 ms
Datos de SGP40	String_1: p = DATACENTRIC + m = SGP40 String_2: p = REQUIRE + m = SGP40	1 microservicio DATACENTRIC	27 ms	21 ms
Microservicio de umbral máximo 30 y mínimo 14	p = RULESERV + r = avg + arg = 30, 14	1 microservicio RULESERV	128 ms	22 ms
Microservicio medio entre A y B	p = RULESERV + r = promedio + arg = A, B	2 microservicios RULESERV	142 ms	69 ms
Datos cada 30 my una regla thrs con un máximo de 31 y un mínimo de 23	String_1: p = DATACENTRIC + f = 30 m String_2: p = RULESERV + f = 30 m + r = thrs + arg = 31, 23	1 microservicio DATACENTRIC + 1 microservicio RULESERV	167 ms	96 ms

** PROBE MATCH significa la cantidad de microservicios que se encuentran en nuestro sistema y que cumplen con la solicitud de la petición PROBE. arg: argumentos, thrs: umbral,prom: promedio.*

Como se muestra en las tablas 2 y 3, aunque el uso de expresiones regulares abre una gran oportunidad para las conexiones en caliente de nuevos dispositivos, el retraso en la interpretación y extracción de datos de estos mensajes es muy alto en comparación con el análisis de mensajes con una estructura común y conocida. Aun así, la utilización de estructuras en el cuerpo del mensaje obliga al par emisor-receptor a utilizar la misma especificación y estructura, y esto no es interesante para microservicios de baja potencia. Al final, se trata de utilizar una configuración u otra según convenga. Por ejemplo, si se trata de microservicios con una baja capacidad computacional, se podría almacenar una petición en lenguaje natural que ocupase mucho menos tamaño que una petición estructurada. También se podría tratar de extender el algoritmo con el objetivo de que sea posible entender otros protocolos de comunicación o lenguajes. Además, con una simple traducción de las listas de palabras asociadas a cada patrón, la funcionalidad de lenguaje natural se puede extrapolar a otros idiomas.

A continuación, presentamos un análisis dedicado a los patrones de entrega datos mediante tiempos del que podremos extraer valiosas conclusiones para decidir en qué momento utilizar un patrón u otro.

6.2. Análisis de latencias del interprete Regexp

Finalmente, analizaremos el impacto que tiene el uso del motor de coreografía y el microservicio de descubrimiento (bróker) en el uso de la CPU y la memoria. Para ello, implementaremos dos máquinas Raspberry Pi que etiquetaremos con los siguientes nombres: RP1 y RP2. El análisis incluye cuatro pasos, que tienen como finalidad analizar la capacidad que una Raspberry Pi puede soportar como máximo:

(1) La Figura 26 (superior) muestra la máquina RP1 que aloja un conjunto de ocho microservicios pero que no intercambian datos entre sí. Los microservicios requieren la mitad de la memoria total de la RP1.

(2) La Figura 26 (en el medio) muestra los ocho microservicios realizando varias tareas de intercambio de datos. Tres microservicios son consumidores de datos y utilizan el patrón solicitud-respuesta; un microservicio es el proveedor de datos de solicitud-respuesta; cuatro microservicios son proveedores centrados en datos. El intercambio de datos es continuo y los mensajes se envían entre un periodo de tiempo de 0,4 y 1,0 s. El uso de la CPU aumenta hasta un 38%.

(3) En la Figura 26 (inferior) se incluye un noveno microservicio, el bróker de descubrimiento. El experimento hace que aumente el consumo de CPU y memoria. El bróker recibe una media de 10 peticiones de descubrimiento por segundo de varios microservicios alojados en RP2. No es necesario conocer el número de microservicios de la máquina RP2, pero sí el número de mensajes que recibimos de esta. La CPU aumenta hasta un 95% y el uso de memoria cerca del 100%. Observamos que, además de los tiempos del proceso de descubrimiento que se muestran en la Tabla 4, el paso de mensajes internos de coreografía introduce una latencia λ de 29,4 ms promedio por petición. El sistema es estable, pero λ varía de 2 a 59 ms con una desviación estándar σ de 18,07 ms en 135 muestras.

(4) Finalmente, se detiene el intercambio de datos entre consumidores y proveedores. Los mensajes de entrada y salida en la RP1 mantienen la actividad de la CPU hasta en un 95%. El experimento reduce λ a 4,97 ms y σ a 2,52 ms en 250 muestras.

Por lo tanto, el impacto de la capa de mensajería de coreografía es variable y λ depende de la actividad general de la máquina, hecho que puede ser un inconveniente en el diseño de

sistemas en tiempo real, aunque la agrupación en clústeres también es factible para limitar la cantidad de microservicios por clúster.

Figura 26 - Uso de CPU y consumo de memoria

En los análisis anteriores hemos focalizado el estudio sobre el sistema de coreografía, pero también consideramos útil realizar un estudio sobre la aplicación móvil para garantizar su correcto funcionamiento en términos de procesamiento de la información y visualización de los datos, dejando de lado la conectividad entre Firebase y la aplicación porque ya existe un gran número de estudios dedicados a esa temática y se trata de una tecnología de conectividad ampliamente probado y testeado.

6.3. Análisis de latencias de aplicación móvil

El objetivo de este apartado es probar y comprobar la aplicación y obtener resultados de estrés que permitan determinar si esta es válida para el uso que se le requiere. Podríamos analizar la capacidad y resistencia que tiene Firebase, el servidor sobre el que se apoya la aplicación, pero esto no tendría sentido porque este servidor tiene un modelo de pago por uso y su capacidad es flexible en función de la inversión económica. Por ello nos centramos en el funcionamiento interno de la aplicación cuando un gran número de datos se deben de mostrar a la vez, dejando de lado cuestiones de latencias entre el servidor y la aplicación.

Nos centraremos en la recogida y visualización de la información. ¿Qué pasará si un sensor comienza a realizar envíos muy frecuentes a la aplicación? ¿Cuánto tardará la aplicación en recoger, limpiar y mostrar esos datos? Para estresar el sistema, se han introducido distintas cantidades de datos y se ha medido el tiempo que tarda el algoritmo de la aplicación en trabajar los datos y mostrarlos. Para medir los tiempos, hemos utilizado la librería StopWatch para Flutter[12] y se han volcado las mediciones en un fichero en el móvil. Además, como la aplicación trabaja sólo con los datos de un día, ha sido necesario borrar los datos de ese día en cada iteración del experimento.

Hemos simulado que un sensor envía datos de humedad a Firebase, utilizando la siguiente función:

$$y = cos(x) + 15$$

Que genera el gráfico que aparece a continuación:

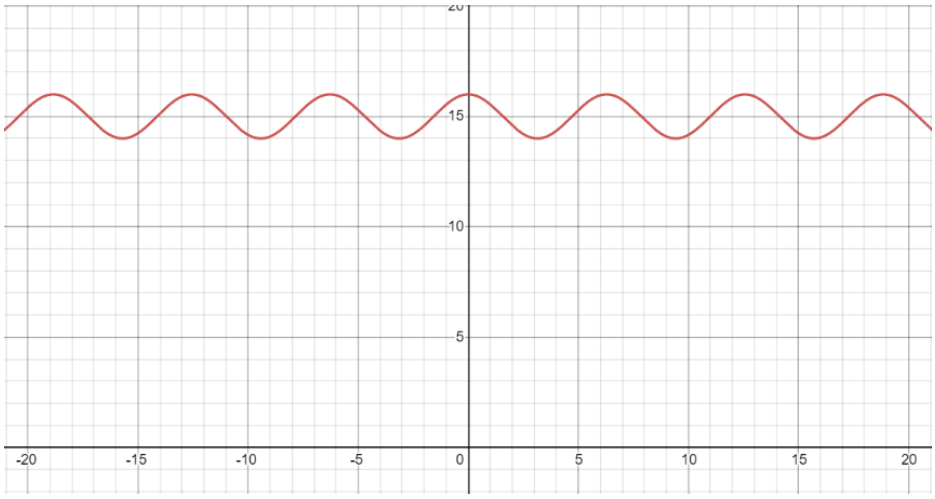

Figura 27. Gráfica de la función y = cos(x) + 15

De esta forma, hemos calculado un número de valores incremental en cada experimento y generado un fichero JSON por cada conjunto de valores. Después, han sido borrados los datos del día del estudio, con el fin de que la aplicación analice sólo los datos que vamos a introducir. A continuación, se muestra el tiempo que ha tardado la aplicación en calcular diez conjuntos de datos distintos e incrementales:

Tabla 4. Relación entre número de valores calculados y tiempo de cálculo

Número de valores	Tiempo de cálculo (ms)
100	7 ms
1000	10 ms
2000	32 ms
5000	73 ms
10000	161 ms
20000	230 ms
50000	922 ms

[12] https://api.dart.dev/stable/2.14.1/dart-core/Stopwatch-class.html

100000	1889 ms
200000	5430 ms

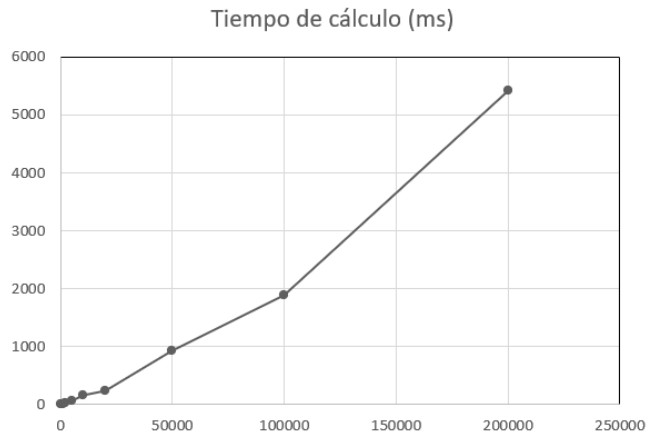

Figura 28 - Tiempo de cálculo de gráficas en función al número de valores.

Observamos que, para una gran cantidad de datos, el tiempo de cálculo que tarda la aplicación es bastante elevado, dejando bloqueada la pantalla del usuario mientras realiza las operaciones. El usuario puede pensar que la aplicación se ha bloqueado, en lugar de que está calculando las gráficas. Por ello, extraemos una característica necesaria para la siguiente versión, que será mostrar una barra de progreso que indique al usuario cuánto le falta para finalizar. Aun así, no se trata de una característica muy importante debido a que actualmente los sensores utilizan la tecnología LoRa y no pueden enviar más de un dato por minuto, teniendo en cuenta el tamaño del dato enviado. Estimando que el sistema tuviese 100 sensores funcionando a tiempo completo, el número máximo de datos que tendría que analizar la aplicación al final del día sería de 144.000 valores analizados. El problema podría surgir si, en el futuro, se permitiese recibir datos de otras tecnologías sin este tipo de limitaciones, como WiFi. En este caso, podríamos recibir un número ilimitado de datos y tendríamos que gestionar el análisis de estos de otra forma.

7. Conclusiones

En este trabajo he aprendido a desarrollar un sistema utilizando diversas tecnologías que son novedad y de gran proyección para mi futuro profesional.

El trabajo ha sido complejo, pero se ha llevado a cabo en el plazo esperado culminando con un prototipo completo. Además, durante el proyecto se han realizado muchos materiales de apoyo para que mi trabajo se pueda utilizar por terceros. Estos materiales no son parte del TFM.

Considero que el desarrollo del proyecto durante este curso ha supuesto un aumento en mi capacidad de comunicación, de análisis y de organización del trabajo. Tanto mi profesora, mentora e impulsora de este proyecto, la Dra. Sara Blanc, como yo estamos realmente satisfechos con el resultado obtenido, en parte por la diversidad de tecnologías que implica y que hemos debido de investigar. En conclusión, considero que este proyecto me ha servido en el desarrollo de mis conocimientos y habilidades y me ha abierto los ojos a campos nuevos de investigación y desarrollo a la par que a descubrir nuevas motivaciones. En mi opinión -a pesar de que me falte experiencia- hemos realizado una toma de decisiones correcta y gracias a eso hemos conseguido completar el proyecto sin grandes contratiempos. Esto ha aumentado mi sentimiento de seguridad hacia el desarrollo de soluciones más complejas en el futuro.

Trabajos futuros

El trabajo realizado para este proyecto ha sido muy extenso porque utiliza muchas tecnologías distintas. Aun así, una solución como la propuesta puede abarcar más aspectos que los comentados en este documento y no se puede tratar todo, aunque se quisiese. Este trabajo se puede ampliar de muchas maneras diferentes, ya sea para adaptarse a las necesidades concretas que se requieran o para cumplir con nuevos estándares y tecnologías. Para ello, simplemente habrá modificar sobre la propuesta o extender la funcionalidad ya existente según sea necesario.

Consideramos que se pueden ampliar varias secciones explicadas en este documento. En primer lugar, sería de gran interés extender el microservicio Regexp, que hasta ahora es capaz de entender una terminología muy específica. Esto podría reducir aún más el tamaño de los mensajes enviados por los microservicios que utilicen este módulo y mejorar la eficiencia en las comunicaciones del sistema. Por otro lado, también sería interesante añadir microservicios que realicen funciones adicionales sobre los datos: por ejemplo, sería interesante programar un microservicio que reciba datos de un tipo y envíe un subconjunto de esos datos de forma ordenada y sin anomalías. También se podrían implementar funciones de filtrado, formateo y algún cálculo sobre los datos sin necesidad de modificar la estructura ya existente, simplemente añadiendo un nuevo microservicio. Otro punto muy interesante para profundizar serían las tecnologías de recepción de los datos de la red de sensores. Estos datos actualmente se reciben a través de un microservicio que controla un hardware adicional a la Raspberry Pi, capaz de recibir datos por LoRa y por WiFi. Sería interesante que nuestro sistema pudiese recibir datos de más tipos de tecnologías, y para ello sólo habría que diseñar un microservicio por cada tecnología y conectarlo con el hardware que necesite para transmitir los datos, en caso de necesitarse. Otra modificación crítica en caso de querer extender el sistema sería eliminar la necesidad de utilizar un bróker centralizado, debido a que se trata de un cuello de botella si tuviera que atender muchas peticiones. Para eliminar este problema se podría diseñar un protocolo de proximidad en el que cada Raspberry Pi disponga de su propio bróker de descubrimiento, y que en caso de no saber resolver una petición se la retransmita a los brókers más cercanos. Por otro lado, también sería interesante crear más patrones de comunicación de los datos, y así aumentar la flexibilidad en las comunicaciones. Por ejemplo, se podrían diseñar patrones de entrega más complejos de los actuales, en los que no sólo intervengan un par de microservicios sino varios más y realicen tareas de forma conjunta. A parte de estas

modificaciones, también sería deseable ampliar mucho más la aplicación móvil, y permitir más formas de visualización de los datos o realizar algún tipo de función sobre el sistema de coreografía, puesto que actualmente sólo es capaz de controlar la electroválvula y de crear alertas. Una posibilidad sería añadir una cámara como sensor y obtener un seguimiento de una variedad cultivada, o implementar un sistema de riego automático que tenga en cuenta más datos como la temperatura, la luminosidad, la humedad del suelo, etc. Esto último se podría hacer sin mayor complicación con la ayuda de algún microservicio que se encargase de recoger la información y tomar decisiones en el sistema de coreografía.

Relación con los estudios cursados

Existe una estrecha relación entre este proyecto y las asignaturas cursadas durante mis estudios universitarios. Para realizar este proyecto hemos tenido que aplicar muchos de los conocimientos técnicos que hemos aprendido durante el Grado y los dos años de Máster, pero también hemos hecho muchas tareas de gestión, organización del trabajo y análisis de posibles soluciones. El desarrollo de estas tareas de gestión ha tenido mucho que ver con asignaturas como “Planificación y Dirección de Proyectos”.

También, el desarrollo de las partes más técnicas ha estado muy ligado con las asignaturas de “Configuración y optimización de sistemas de cómputo”, “Ciberseguridad” y “Redes y Seguridad” debido a la necesidad de mantener un sistema distribuido de forma segura y eficiente. Por otro lado, la existencia de asignaturas como “Sistemas Inteligentes” ha facilitado el desarrollo de los microservicios inteligentes y de los patrones de entrega de datos, que tienen como objetivo cubrir una gran cantidad de escenarios posibles de comunicación. Además, el despliegue de la base de datos ha sido mucho más fácil debido a los conocimientos aprendidos en asignaturas como “Redes y Seguridad” y “Bases de datos”.

La asignatura de “Computación de altas prestaciones” ha ayudado a organizar el trabajo y diseñar funcionalidades que se pueden distribuir, y a mantener una serie de buenas prácticas durante el desarrollo del bróker de descubrimiento. Por otro lado, también se han aplicado conocimientos de las asignaturas de “Desarrollo de videojuegos” y “Aplicaciones gráficas y multimedia” que, aunque no parezcan estar relacionadas, han ayudado mucho en el diseño de la aplicación móvil que debía ser llamativa y fácil de utilizar por un niño. Por último, consideramos que existe también una amplia relación con la asignatura de “Sistemas empotrados y ubicuos” ya que nuestra solución utiliza Raspberry pi que tienen características similares a los sistemas empotrados.

Relación con competencias específicas

Más allá de la aplicación de los distintos conocimientos adquiridos a lo largo de los estudios de Grado y Máster, el desarrollo del presente proyecto nos ha servido para alcanzar las competencias específicas establecidas por la UPV para los futuros titulados. Por una parte, creemos que hemos demostrado que somos capaces de comprender y aplicar conocimientos avanzados de computación y de algoritmia a problemas de ingeniería.
Desde nuestro punto de vista, en este proyecto hemos realizado una planificación estratégica y una elaboración adecuada, atentos a los criterios de calidad necesarios. Creemos que con el desarrollo del proyecto hemos demostrado nuestra capacidad para conceptualizar, diseñar, desarrollar y evaluar la interacción persona-ordenador de productos, sistemas, aplicaciones y microservicios informáticos, así como para analizar las necesidades de información que se plantean en un entorno y llevar a cabo en todas sus etapas el proceso de construcción de un sistema de información en un entorno multidiciplinar. Asimismo, al llevarlo a término demostramos nuestra competencia para asegurar, gestionar, auditar y certificar la calidad de los desarrollos, procesos, sistemas, microservicios, aplicaciones y productos informáticos sin dejar a un lado la importancia de desarrollar, gestionar y evaluar mecanismos de certificación y

garantía de seguridad en el tratamiento y acceso a la información en un sistema de procesamiento local o distribuido.
Por otra parte, no solo es esencial para un titulado la aptitud para la aplicación e integración de los conocimientos adquiridos y para resolver problemas en entornos nuevos o poco conocidos dentro de contextos multidisciplinares, sino que no hay que desestimar la importancia de las habilidades comunicativas a la hora de presentar y defender proyectos y conclusiones ante distintos públicos de un modo claro y sin ambigüedades, destreza que será esencial en el posterior desarrollo de nuestras actividades profesiones.

Pero los estudios realizados, más allá del desarrollo de unas competencias específicas que atañen a todos los futuros ingenieros informáticos formados en la UPV, nos han hecho alcanzar otras de carácter trasversal, que son la base de la filosofía educativa de esta Universidad, a saber: la aplicación de los conocimientos y el pensamiento práctico a la par que crítico; la importancia de la innovación, de la creatividad y del emprendimiento, siempre a partir del conocimiento de los problemas actuales en nuestro campo de especialización y en otros campos del conocimiento; la capacidad de trabajo en equipo y liderazgo; la responsabilidad ética, profesional y medioambiental. Es importante para todos nosotros que el aprendizaje no acabe cuando nos entregan el título, sino que debemos dedicarnos permanentemente al aprendizaje.

8. Referencias

1. *A tool chain for choreographic design.* **Guanciale, Roberto and Tuosto, Emilio. 2020.** September 2020, Science of Computer Programming, Vol. 202, p. 102535.

2. *A comparison of AMQP and MQTT protocols for Internet of Things.* **Uy, Nguyen Quoc and Nam, Vu Hoai. 2019.** 2019. 2019 6th NAFOSTED Conference on Information and Computer Science (NICS). pp. 292-297.

3. *A Service Discovery Solution for Edge Choreography-Based Distributed Embedded Systems.* **Blanc, Sara, et al. 2021.** January 2021, Sensors, Vol. 21, p. 672.

4. *A Smart Decision System for Digital Farming.* **Cambra Baseca, Carlos, et al. 2019.** 2019, Agronomy, Vol. 9. ISSN: 2073-4395.

5. *A Study of LoRa: Long Range & Low Power Networks for the Internet of Things.* **Augustin, Aloÿs, et al. 2016.** 2016, Sensors, Vol. 16. ISSN: 1424-8220.

6. *A Universal Complex Event Processing Mechanism Based on Edge Computing for Internet of Things Real-Time Monitoring.* **Lan, Lina, et al. 2019.** July 2019, IEEE Access, Vol. PP, pp. 1-1.

7. *An Internet of Things (IoT) Architecture for Smart Agriculture.* **Verma, Saurav, et al. 2018.** 2018. 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA). pp. 1-4.

8. *Automated synthesis of mediators for middleware-layer protocol interoperability in the IoT.* **Bouloukakis, Georgios, et al. 2019.** s.l. : Elsevier, December 2019, Future Generation Computer Systems, Vol. 101, pp. 1271-1294.

9. **Boukharouba, Nour. 2020.** Desarrollo de un sistema para el control de cultivos para agricultura de precisión basado en IoT. 2020.

10. *Choreographing Services for Smart Cities: Smart Traffic Demonstration.* **Chen, Lei and Englund, Cristofer. 2017.** 2017. pp. 1-5.

11. *CHOReVOLUTION: Service choreography in practice.* **Autili, Marco, et al. 2020.** June 2020, Science of Computer Programming, Vol. 197, p. 102498.

12. *ChorSystem: A Message-Based System for the Life Cycle Management of Choreographies.* **Weiß, Andreas, et al. 2016.** [ed.] Christophe Debruyne, et al. Cham : Springer International Publishing, 2016. On the Move to Meaningful Internet Systems: OTM 2016 Conferences. pp. 503–521. ISBN: 978-3-319-48472-3.

13. *Complementing IoT Services Through Software Defined Networking and Edge Computing: A Comprehensive Survey.* **Rafique, Wajid, et al. 2020.** 2020, IEEE Communications Surveys Tutorials, Vol. 22, pp. 1761-1804.

14. **Coto-Santiesteban, Alex, Guanciale, Roberto and Tuosto, Emilio. 2020.** Choreographic Development of Message-Passing Applications: A Tutorial. 2020, pp. 20-36.

15. **Cruz-Filipe, Luís, Montesi, Fabrizio and Safina, Larisa. 2019.** Implementing choreography extraction. *Implementing choreography extraction.* October 2019.

16. *Design, Resource Management, and Evaluation of Fog Computing Systems: A Survey.* **Martinez, Ismael, Hafid, Abdelhakim Senhaji and Jarray, Abdallah. 2021.** 2021, IEEE Internet of Things Journal, Vol. 8, pp. 2494-2516.

17. *Dynamic IoT Choreographies.* **Seeger, Jan, et al. 2019.** January 2019, IEEE Pervasive Computing, Vol. 18, pp. 19-27. ISSN: 1558-2590.

18. *Evaluating IoT service composition mechanisms for the scalability of IoT systems.* **Arellanes, Damian and Lau, Kung-Kiu. 2020.** March 2020, Future Generation Computer Systems, Vol. 108.

19. *Exploiting smart e-Health gateways at the edge of healthcare Internet-of-Things: A fog computing approach.* **Rahmani, A., et al. 2018.** 2018, Future Gener. Comput. Syst., Vol. 78, pp. 641-658.

20. *FIPA@ Towards a Standard for Software Agents.* **O'Brien, P. and Nicol, R. 1998.** 1998, BT Technology Journal, Vol. 16, pp. 51-59.

21. **Hahn, Michael, et al. 2017.** *TraDE - A Transparent Data Exchange Middleware for Service Choreographie.* 2017.

22. **Hollosy, Thomas.** Actorsphere. *Actorsphere.*

23. *Improving Health Monitoring With Adaptive Data Movement in Fog Computing.* **Cappiello, C., et al. 2020.** 2020, Frontiers in Robotics and AI, Vol. 7.

24. *Integration of Distributed Services and Hybrid Models Based on Process Choreography to Predict and Detect Type 2 Diabetes. Sensors.* **Singh, K., et al. 2017.** [ed.] A. Heidari P. Allahverdizadeh M. A. Jabraeil Jamali and F. Norouzi. Palladam, India; Khalde, Lebanon; Eger, Hungary; Thesssaloniki, Greece; Rome, Italy; Valencia, Spain; Bayo-Montón; Auckland, New Zealand; Nature; Cham, Switzerland; Cham, Switzerland; Paris-Rocquencourt, France; Antwerp, Belgium; Cham, Switzerland; Cardiff, United Kingdom; Cham, Switzerland; Cham, Switzerland; Stockholm, Sweden; Sydney, Australia, 4-7; Cambridge, MA, USA; Cambridge, MA, USA; Fremont, CA, USA; Monastir, Tunisia; Armonk, NY, USA; Hanoi, Vietnam; Negash, B; New York, NY, USA; Chennai, India; Burlington, MA, USA; San Francisco, CA, USA; Zhangjiajie, China : Springer International Publishing, 2017, A Tool Chain for Choreographic design. Sci. Comput. Program, Vol. 20, pp. 9–31.

25. *Intelligent Agriculture and Its Key Technologies Based on Internet of Things Architecture.* **Chen, Jinyu and Yang, Ao. 2019.** 2019, IEEE Access, Vol. 7, pp. 77134-77141.

26. *Internet of Things (IOT): An overview and its applications.* **Dudhe, P. V., et al. 2017.** 2017. 2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS). pp. 2650-2653.

27. *Internet of Things: Survey and open issues of MQTT Protocol.* **Bani Yassein, Muneer, et al. 2017.** 2017.

28. **ISO.** Internet of Things (loT) — Reference Architecture. *Internet of Things (loT) — Reference Architecture.* [Online] https://www.iso.org/obp/ui/#iso:std:iso-iec:30141:ed-1:v1:en.

29. **Jabraeil Jamali, Mohammad, et al. 2019.** *Towards the Internet of Things: Architectures, Security, and Applications.* 2019. ISBN: 978-3-030-18468-1.

30. *Middleware for Smart Gateways Connecting Sensornets to the Internet.* **Bimschas, Daniel, et al. 2010.** 2010.

31. *Modeling Data Transformations in Data-Aware Service Choreographies.* **Hahn, Michael, et al. 2018.** 2018. 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC). pp. 28-34.

32. **OASIS.** MQTT for Sensor Networks (MQTT-SN). *MQTT for Sensor Networks (MQTT-SN).* [Online] (Accessed on 09/12/2021). https://www.oasis-open.org/committees/download.php/66091/MQTT-SN_spec_v1.2.pdf.

33. **SABIEN. 2021.** http://www.sabien.upv.es/. *http://www.sabien.upv.es/.* [Online] 2021. http://www.sabien.upv.es/.

34. **Savanovic, Zorica, Galletta, Letterio and Vieira, Hugo. 2020.** A type language for message passing component-based systems. *A type language for message passing component-based systems.* September 2020.

35. *Semantic Process Choreography for Distributed Sensor Management.* **Fernandez-Llatas, Carlos, et al. 2010.** 2010. pp. 32-37.

36. **Shelby, Zach, Hartke, Klaus and Bormann, Carsten. 2014.** The Constrained Application Protocol (CoAP). *The Constrained Application Protocol (CoAP).* s.l. : RFC Editor, June 2014.

37. *Survey of platforms for massive IoT.* **Hejazi, Hamdan, et al. 2018.** 2018. pp. 1-8.

38. **W3C.** SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). [Online] (Accessed on 09/12/2021). https://www.w3.org/TR/soap12-part1/.

39. *Wearable Sensors Integrated with Internet of Things for Advancing eHealth Care.* **Bayo Montón, José, et al. 2018.** June 2018, Sensors, Vol. 18, p. 1851.

40. *Web Services Orchestration and Choreography.* **Peltz, C. 2003.** 2003, Computer, Vol. 36, pp. 46-52.

41. *What is Edge computing?* **Cisco.**

9. Anexo I

¿Cómo se define cada tipo de microservicio?

A continuación, se presentan varios tipos de mensajes HELLO, uno por cada modelo de comunicación y otro último que describe dos modelos de comunicación a la vez. Es importante mencionar que un servicio puede publicarse en cualquiera de los modelos o incluso en varios a la vez.

1. Mensaje HELLO de un servicio REQ-RES

XFIPAMSG {

<d:Hello>

**Dirección en coreografo

<a:EndpointReference>

<a:Address>Communication.HumidityService**</a:Address>**

</a:EndpointReference>

** Tipo de servicio

[<d:Types>i:humidity **</d:Types>]?**

**Alcance del servicio

<d:Scopes>I am a req-res service**</d:Scopes>**

**Otros puntos de acceso

[<d:XAddrs>list of xs:anyURI**</d:XAddrs>]?**

<d:MetadataVersion>0000**</d:MetadataVersion>**

</d:Hello>

2. Mensaje HELLO de un servicio DATACENTRIC

XFIPAMSG {

<d:Hello>

**Dirección en coreografo

<a:EndpointReference>

<a:Address>Communication.HumidityService**</a:Address>**

</a:EndpointReference>

** Tipo de servicio

[<d:Types>i:humidity **</d:Types>]?**

**Alcance del servicio

<d:Scopes>Send data every 30 min from STM32 sensor**</d:Scopes>**

**Otros puntos de acceso

[<d:XAddrs>list of xs:anyURI**</d:XAddrs>]?**

<d:MetadataVersion>0000**</d:MetadataVersion>**

3. Mensaje HELLO de un servicio RULES-SERVER

XFIPAMSG {

<d:Hello>

**Dirección en coreografo

<a:EndpointReference>

<a:Address>Communication.HumidityService**</a:Address>**

</a:EndpointReference>

** Tipo de servicio

[<d:Types>i:humidity **</d:Types>]?**

**Alcance del servicio

<d:Scopes> Accept rules of average and threshold **</d:Scopes>**

**Otros puntos de acceso

[<d:XAddrs> list of xs:anyURI **</d:XAddrs>]?**

<d:MetadataVersion>0000**</d:MetadataVersion>**

4. Mensaje HELLO de un servicio de tipos RULES-SERVER y DATACENTRIC

```
XFIPAMSG {

 <d:Hello>
        **Dirección en coreografo
        <a:EndpointReference>

        <a:Address>Communication.HumidityService</a:Address
>
        </a:EndpointReference>
        ** Tipo de servicio
        [<d:Types>i:humidity </d:Types>]?
        **Alcance del servicio
        <d:Scopes>Accept rules of average and send data from
        STM32 Sensor every 30 minutes</d:Scopes>
        **Otros puntos de acceso
        [<d:XAddrs>list of xs:anyURI</d:XAddrs>]?

        <d:MetadataVersion>0000</d:MetadataVersion>
```

Estructura interna de los mensajes PROBE y PROBE-MATCH RESOLVE y RESOLVE-MATCH

La estructura interna de los mensajes que participan en el descubrimiento de nuevos servicios es la siguiente para cada tipo de mensaje:

PROBE

Este mensaje comprende en su interior el tipo de medidas de sensor que desea recibir, la frecuencia mínima a la que se desea suscribirse y la forma en la que se recibirán esos datos. Esta información se expresará utilizando expresiones regulares e irá dirigida únicamente al bróker. A continuación se observa la estructura interna de un mensaje Probe:

XFIPAMSG {

<d:Probe>
** Tipo de servicio
[<d:Types>i:humidity **</d:Types>]?**
**Alcance del servicio
<d:Scopes>Need service of REQ-RES and need to receive data from STM32 Sensor every 30 minutes**</d:Scopes>**
</d:Probe>

}

PROBE-MATCH

El bróker recibe el mensaje anterior y analiza la petición contenida en él. Mediante el análisis regexp obtiene los servicios 'target' que cumplen con la especificación del solicitante. El bróker, después, compone un mensaje de tipo *ProbeMatch* con los servicios que han cumplido con los requisitos y se lo envía al servicio que comenzó la comunicación.

```
XFIPAMSG {

 <d:ProbeMatches>
    <d:ProbeMatch>
        **Dirección en coreografo
        <a:EndpointReference>

        <a:Address>Communication.humidityService</a:Address
>
        </a:EndpointReference>
        ** Tipo de servicio
        [<d:Types>i:humidity </d:Types>]?
        **Alcance del servicio
        <d:Scopes> I am a req-res service </d:Scopes>

        [<d:XAddrs>list of xs:anyURI</d:XAddrs>]?

        <d:MetadataVersion>0001</d:MetadataVersion>
    </d:ProbeMatch>
 ....
    <d:ProbeMatch>
        ....
    </d:ProbeMatch>

 </d:ProbeMatches>

}
```

RESOLVE

Posteriormente, el servicio que inició la comunicación con el bróker, que ya ha recibido el mensaje PROBE-MATCH, reenvía los un subconjunto de los servicios recibidos en el probe-match utilizando el mensaje RESOLVE.

XFIPAMSG {

<d:Resolves>
<d:Resolve>
<a:EndpointReference>

<a:Address>Communication.humidityService**</a:Address>**
</a:EndpointReference>
** Tipo de servicio
[<d:Types>i:humidity **</d:Types>]?**
**Alcance del servicio
<d:Scopes>Need service of REQ-RES and need to receive data from STM32 Sensor every 30 minutes**</d:Scopes>**

</d: Resolve>

RESOLVE-MATCH

Por último, el mensaje RESOLVE-MATCH lo envía el bróker cuando acabó de realizar el registro del servicio. En caso de que en el mensaje RESOLVE hubiese varios servicios, el mensaje RESOLVE-MATCH indica al cliente a qué servicio(s) se ha registrado y cómo debe comenzar a comunicarse con él/ellos. En este proceso se aplica un balanceo de la carga para asignar al servicio solicitante el servicio o servicios con menor número de clientes o con menos trabajo.

XFIPAMSG {

<d:ResolveMatches>
<d:ResolveMatch>
<a:EndpointReference>

<a:Address>Communication.humidityService**</a:Address>**
</a:EndpointReference>
** Tipo de servicio
[<d:Types>i:humidity **</d:Types>]?**
**Alcance del servicio
<d:Scopes>REQ-RES service**</d:Scopes>**

</d: ResolveMatch>

…

En caso de que se haya solicitado un servicio de tipo DATACENTRIC, el bróker deberá facilitar al cliente una máscara a la que se pueda suscribir para recibir la información enviada por el servicio DATACENTRIC. Esa máscara la recibe a través del mensaje RESPONSE-MATCH, en el campo *Scopes*.

Ejemplo de comunicación REQUEST-RESPONSE

El tipo de patrón de datos del mensaje se especifica en un campo de la cabecera del mensaje XFIPA. El modelo REQ-RES implica que un servicio realice una solicitud y otro le responda. Para ello, el servicio solicitante compone la cabecera del mensaje de la siguiente forma:

```
{
 Sender: Communication.StartService <Identificador de servicio origen>
 Receiver: Communication.ReqResService1 <Identificador de servicio
destino>
 Type: REQ-RES
 Body: {
        ... Estructura WS-Discovery
        }
```

Un proveedor de datos que atiende el patrón REQ-RES recibe este mensaje, procesa la petición y responde inmediatamente con un mensaje como el siguiente:

Para ello almacena la identificación del servicio solicitante que está en el parámetro *Sender* y cuando finalice el proceso de recogida de la información le responderá.

Ejemplo de comunicación DATACENTRIC

El más simple de todos. Un emisor envía datos sobre una máscara que será la que previamente el bróker ha facilitado al receptor de los datos. El receptor, que ya está suscrito a esa máscara, recibirá los mensajes enviados utilizando esta máscara. En este patrón solo interviene un mensaje; el que envía periódicamente el servicio publicador.

```
{
 Sender: Communication.DataCentricService1 <Identificador de servicio
origen>
 Receiver: humidityData <máscara de recepción>
 Type: DATACENTRIC
 Body: {
        ... Estructura WS-Discovery
        }
```

El campo *Sender* contiene la dirección del servicio origen el mensaje. El campo *Receiver* en este caso está compuesto por la misma máscara de recepción a la que los servicios solicitantes se han suscrito. El campo *Type* describe el patrón de comunicación y el campo *Body* contiene el mensaje con la estructura de WS-Discovery vista anteriormente.

Ejemplo de comunicación RULES-SERVER

El patrón Rules-server utiliza también un único mensaje que se envía cuando se cumple la regla en el proveedor. La regla viene determinada en uno de los campos para que el cliente sepa qué regla es la que se ha cumplido, porque puede haber varias para un mismo proveedor.

{
Sender: Communication.DataCentricService1 <Identificador de servicio origen>
Receiver: humidityData <máscara de recepción>
Type: RULES-SERVER
Rule: { type: “avg”, args:”50”}
Body: {
... Estructura WS-Discovery
}

10. Anexo II

Para realizar el despliegue del sistema, se deberán realizar las siguientes tareas en el orden descrito:

1. Conectar los sensores requeridos a los Arduinos y etiquetar cada Arduino con el tipo de datos que va a enviar y la parcela a la que pertenece.
2. Instalar Windows IOT en cada Raspberry Pi. Microsoft ofrece una aplicación a modo de panel de control, "IoT Dashboard", que instala el sistema operativo sobre una Raspberry conectada al mismo ordenador donde está el panel de control. Esta aplicación se puede encontrar en Microsoft Store, y se debe descargar sobre un ordenador Windows.

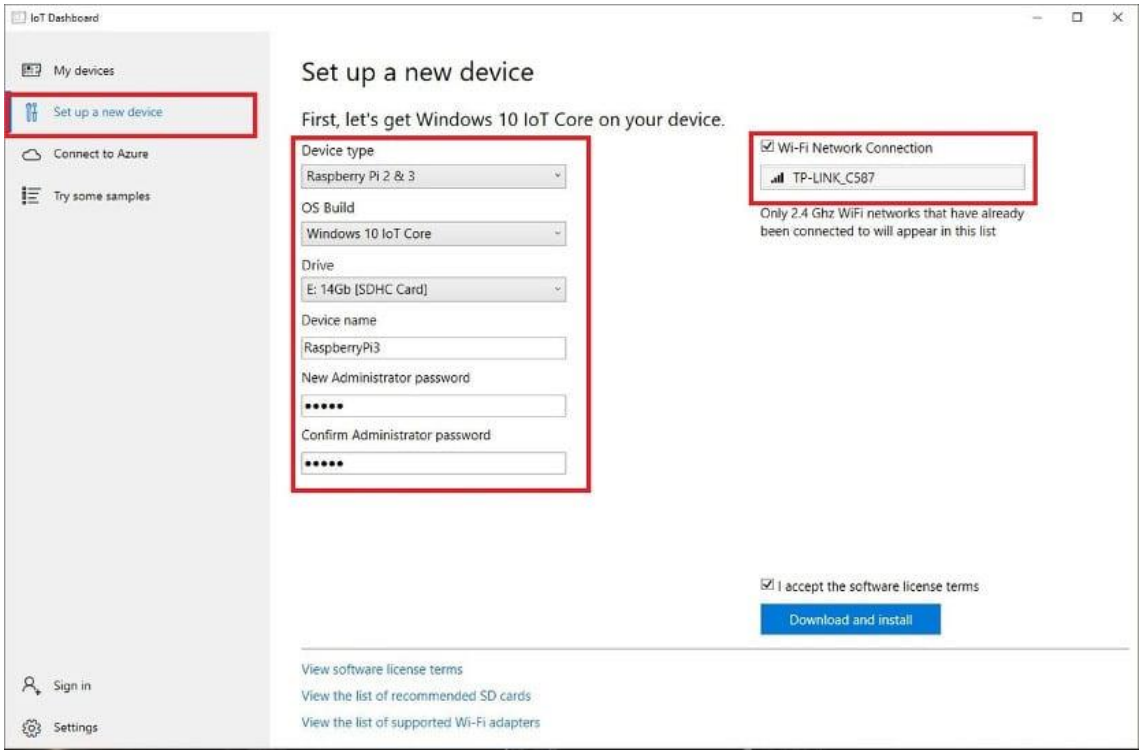

Ilustración 1. Pantalla de instalación de Windows IoT en el panel de control

3. Habrá que conectar la Raspberry Pi al sistema anfitrión y el panel de control detectará el dispositivo.. Esto no implicara ninguna dificultad aunque el usuario carezca de formación informática. Posteriormente se debe apretar al botón "Download and install" y el proceso comenzará automáticamente. Este proceso habrá que repetirlo para cada Raspberry. Cuando finalice, ya tendremos Windows IoT instalado en nuestras máquinas.
4. El siguiente paso es instalar el sistema de coreografía en cada Raspberry Pi y añadir los microservicios. Aunque este es el punto más complejo, se facilita una plantilla con los tres microservicios del esquema para que el usuario pueda ejecutar directamente el sistema sin configurar los microservicios, únicamente tiene que ubicar las máquinas en la misma red local, ya sea por cable o mediante una conexión WiFi. Para instalar el sistema, los profesores deben tener acceso de administrador vía SSH y ejecutar un *script* que descarga el proyecto del repositorio y lo instala.

5. Será necesario crear un servidor de Firebase con la configuración por defecto. En nuestro caso, habrá que crear un proyecto de tipo *RealTime DB*, que es una de las configuraciones por defecto de Firebase.
6. Desde Firebase, crear una aplicación asociada al proyecto (Sección 1 de la ilustración inferior) y descargar el fichero "Google-services.json" (Sección 2).

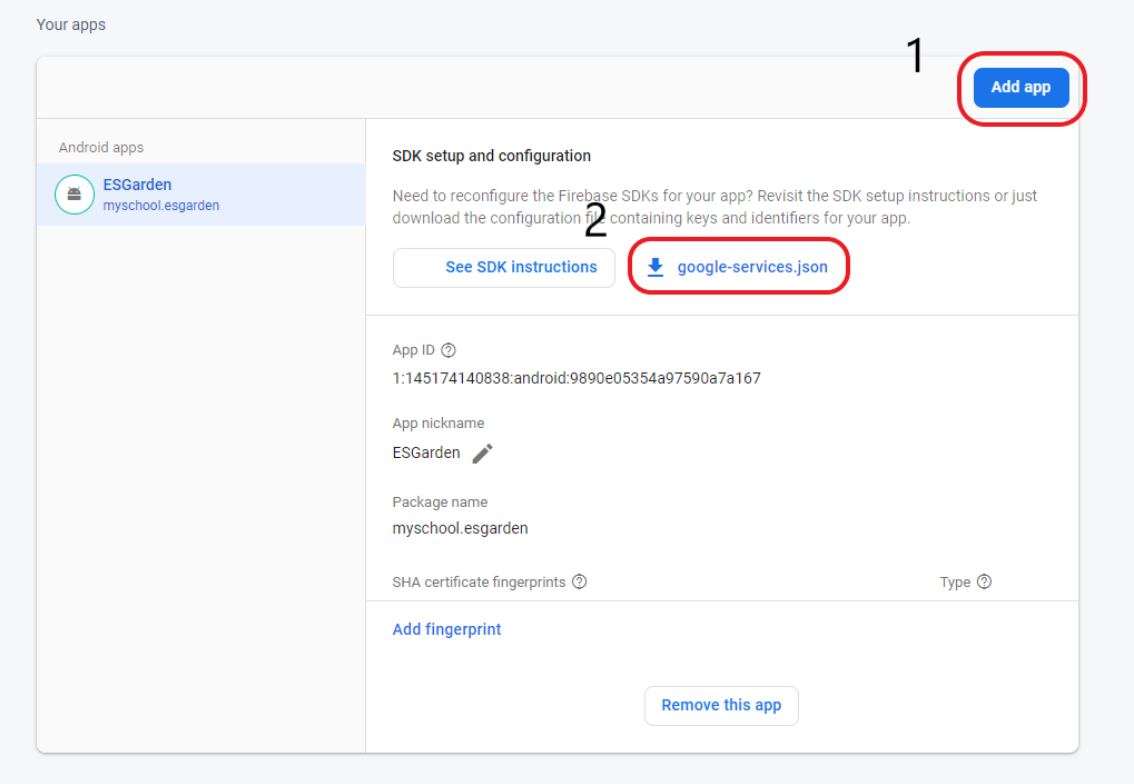

Ilustración 2 . Configurar aplicación en Firebase

7. A continuación, se procederá a descargar el proyecto de la aplicación del repositorio oficial (https://github.com/senenpalanca/esgarden) y abrirlo desde *Android Studio*. Para poder abrir correctamente el proyecto, se necesitarán los plugins de Dart y Flutter, que se instalarán desde la configuración de Android studio, como se muestra en la siguiente ilustración.

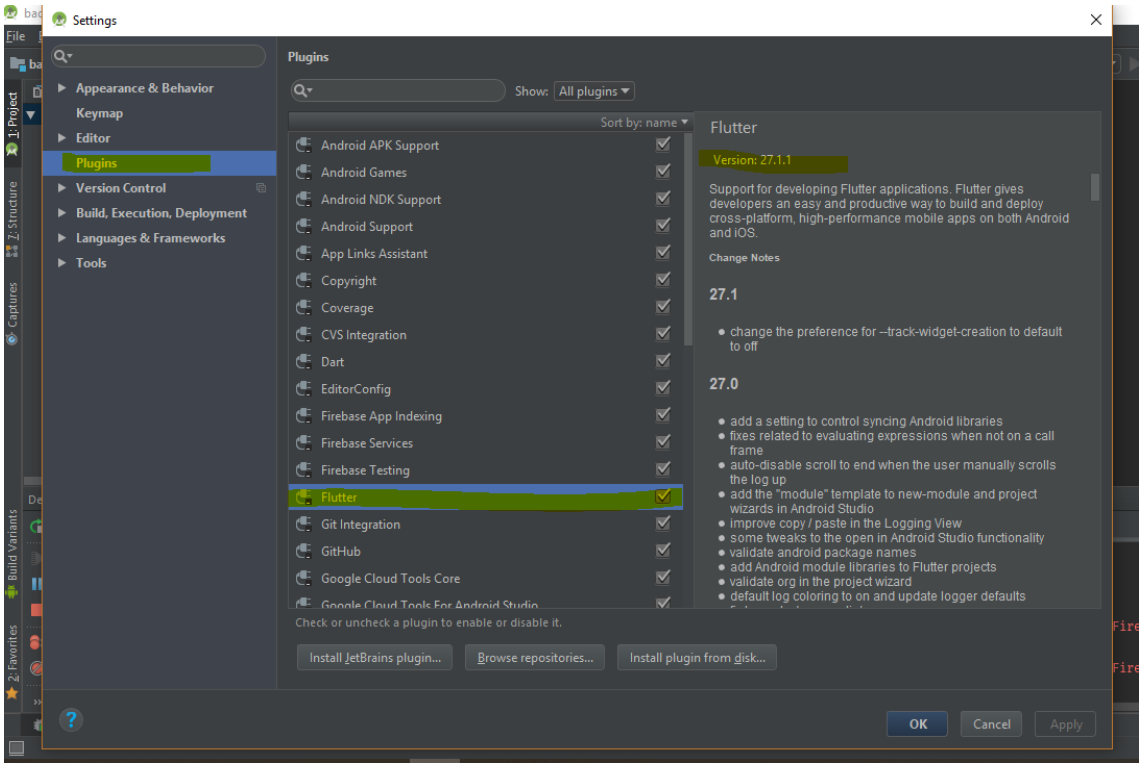

Ilustración 1. Ajustes de Android Studio. Instalación de Plugins

Deberemos ubicar el fichero descargado en el paso anterior en la carpeta "/app" del proyecto antes de ejecutarlo. Aquí se observa dónde se debe dejar el archivo:

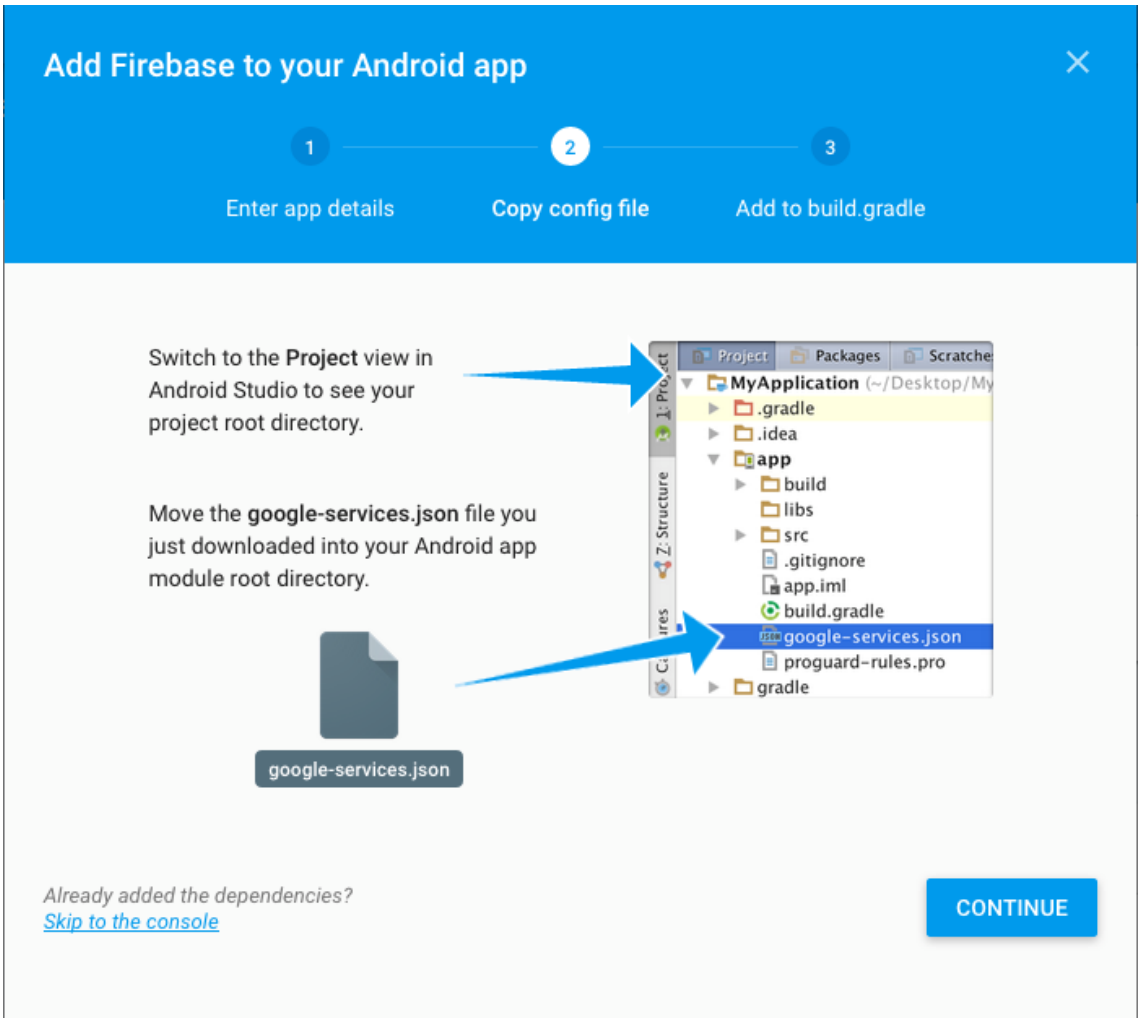

Ilustración 3. Añadir fichero google-services.json al proyecto de la App

8. Compilar y ejecutar el proyecto de la aplicación en Android Studio para visualizar los datos del sistema.

A partir de este punto ya se contará con el sistema desplegado. Para verificar que este funciona correctamente,se pondrá en marcha uno de los sensores y habrá que comprobar que los datos se reciben y muestran en la aplicación correctamente.

En caso de querer ampliar el sistema, habrá que analizar qué parte del sistema es la que se desea extender. Es posible tratar una gran cantidad de sensores por cada Raspberry, por lo que no será necesario extender la sección de coreografía a menos que se requiera más potencia en el sistema distribuido. La sección de la aplicación simplemente necesita que la parte del servidor de datos (Firebase) reciba datos de forma concurrente. Además, si se desea ampliar el sistema de coreografía, bastará con ejecutar el proyecto de plantilla en una tercera Raspberry Pi, que se conectará a las otras dos de forma transparente.

En caso de que haya dudas o para obtener información más detallada de todo el proceso, se facilitan los siguientes tutoriales. En ellos se explica el proceso de despliegue de forma visual e interactiva.

- Añadir el flutter de plugin
  https://youtu.be/j-BN7-YXogo

- Descargar el proyeto del repositorio
  https://youtu.be/qVgn89lLxcY

- Importar el proyecto en Android studio
  https://youtu.be/yaIW2MRl3ss

- Abrir proyecto en Android Studio, compilarlo por primera vez y solucionar errores por falta de librerías
  https://youtu.be/1Rlv1PDvbe4

- Conectar base de datos Firebase con el proyecto de Android studio
  https://youtu.be/zVfxL0Uv2vU

- Ejecutar aplicación en el emulador de Android studio
  https://youtu.be/wMoIQ9mPZN4

- Cambiar el nombre de un paquete en Android studio (Necesario para subir app a PlayStore)
  https://youtu.be/WhFgP2yRgUY