PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

Diseño e implementación de un *middleware* para la integración horizontal de aplicaciones y dispositivos IoT usando la arquitectura de microservicios

TESIS PARA OBTENER EL TÍTULO PROFESIONAL DE INGENIERO DE LAS TELECOMUNICACIONES

AUTOR:
Alejandro Macedo Pereira

ASESOR:
Jorge Benavides Aspiazu

Junio, 2020

Resumen

El presente trabajo consiste en el diseño e implementación de un *middleware* para la integración horizontal de aplicaciones y dispositivos IoT. Este problema nace por la heterogeneidad de los diversos componentes de los dispositivos entre los que se encuentran los distintos tipos de sensores y las mediciones que realizan, los diversos protocolos de comunicación que existen y la flexibilidad con la que se cuenta para combinar entre los casos anteriores. Por lo tanto, se requiere de un sistema que sea capaz de procesar dispositivos de diversas características y protocolos de comunicación para convertirla en un formato mucho estandarizado y que pueda ser usado fácilmente por desarrolladores con pocos conocimientos específicos de IoT. Ese sistema estará desarrollado bajo una arquitectura de microservicios, lo que permite ser fácilmente extensible y desarrollarse en el lenguaje que se desee, implementándose en este trabajo seis servicios que permiten registrar y almacenar la información enviada por diversos dispositivos IoT separando el acceso a los recursos mediante un sistema de usuarios y autenticación.

En el primer capítulo se describe la formulación del diseño teórico del sistema, desde la problemática, los objetivos y finalmente los requerimientos establecidos para la implementación.

El segundo capítulo es un estudio del estado del arte de *middlewares* de integración vertical, donde se estudian estándares propuestos y dos implementaciones para la solución de este problema.

El tercer capítulo muestra el diseño y desarrollo final del sistema, explicándose el flujo de funcionamiento de la información de los dispositivos y cada uno de los servicios implementados. También se explica el modelo de despliegue usado.

En los capítulos finales se presenta el modelo de pruebas desarrollados y se estudian los resultados obtenidos.

Finalmente, se describen las conclusiones obtenidas del diseño y la implementación del sistema, así como las recomendaciones para posibles trabajos futuros.

Índice

Índice de Figuras

Índice de Tablas

Introducción

A día de hoy, las redes IoT se hacen presente en diferentes escenarios no solo domésticos, sino también industriales y productivos. Una de las principales dificultades de implementaciones IoT es que principalmente se usan modelos de integración vertical, donde para el desarrollo de un servicio es necesario contar con la arquitectura completa de configuración y despliegue de sensores, conexión, comunicación, procesamiento de los datos y la aplicación. Esto ocurre por la gran variedad de dispositivos, sensores, protocolos de comunicación y métodos de conexión con una red de datos para los dispositivos, lo que genera que cada uno de los servicios se encuentre muy optimizado para la función que realice, pero lo convierte, al mismo tiempo, en muy inflexible a integrarse con otros servicios ya presentes.

La presente tesis usa el modelo de abstracción de dispositivos IoT propuesto en la tesis de posgrado del asesor, la cual será ampliada e implementada para su funcionamiento en una arquitectura de microservicios. Por lo tanto, la presente tesis tiene como objetivo el diseño de un middleware con un modelo de abstracción que englobe a dispositivos IoT sin importar el tipo de protocolo, las características o tipo de información que sensen; lo que permitirá a cualquier usuario usar la información recolectada por sus dispositivos a través de una interfaz de datos estándar que podrá ser usada en cualquier tipo de aplicación. Además, este sistema será diseñado usando la arquitectura de microservicios, lo que permitirá que sea escalable de acuerdo a la demanda de los dispositivos y usuarios, permitiendo también que sea fácilmente extensible si se requiere alguna funcionalidad adicional.

Capítulo 1. Formulación del diseño del middleware y sus requerimientos

El presente capítulo describe la justificación del problema, se definen los objetivos, las razones que motivaron el trabajo y se formula el diseño del *middleware*.

1.1. Definición del problema y justificación

1.1.1. Internet de las Cosas (IoT)

El Internet de las Cosas (o IoT por sus siglas en inglés, *Internet of Things*) es una red compuesta por diferentes objetos, que pueden ser artículos domésticos cotidianos como lámparas, focos, refrigeradoras o vehículos y herramientas industriales, que se conectan a una red da datos, generalmente Internet, para intercambiar la data que adquieren a través de diversos sensores y software embebido [1]. Según [1], IoT comenzó a tener una gran explosión de uso en los últimos años debido a diversos factores, entre los que se encuentran:

- Acceso a sensores de bajo costo y bajo consumo de energía, lo que permite que se puedan construir dispositivos IoT a gran escala y económicamente viables.

- Mejores protocolos de comunicación que permiten transmisiones más eficientes y en peores condiciones.

- Plataformas de computación en la nube que permiten el acceso casi infinito a la infraestructura necesaria para trabajar con el volumen de información recolectado.

- El avance de *machine learning* y *analytics* que permite la extracción de la información de forma más eficiente y sobre más volumen de datos, ayudando a las empresas en su toma de decisiones y análisis del entorno sobre el que trabajan.

Por ejemplo, se usan sensores en suelos de granjas que miden el nivel de humedad en un radio cercano de donde se encuentran instaladas. Estas envían periódicamente sus mediciones a servidores que notifican al granjero si un parche se encuentra deshidratado comparando su nivel de humedad con información del tipo de planta que se encuentra sembrado. Además, se podría automatizar aún más el proceso e implementar máquinas que automáticamente rieguen solo los parches de suelo que sean necesarios y sumado a información meteorológica de lluvias consigan un ahorro significativo en el uso de agua a los granjeros. Aparte podrían usarse otro tipo de sensores que midan los niveles de luz, calidad de aire, temperatura, etc. y compararlo con los datos de cosechas pasadas o que se encuentren en otras partes del mundo para brindar la información necesaria que permita que las plantas crezcan de forma óptima [2].

El número de dispositivos IoT ha aumentado considerablemente, con aproximadamente 26.66 mil millones para el año 2019 y con predicciones que indican que esta cifra aumentará a 75.44 mil millones para el año 2025 y que se beneficiará de nuevas tecnologías de conexiones inalámbricas como 5G [3].

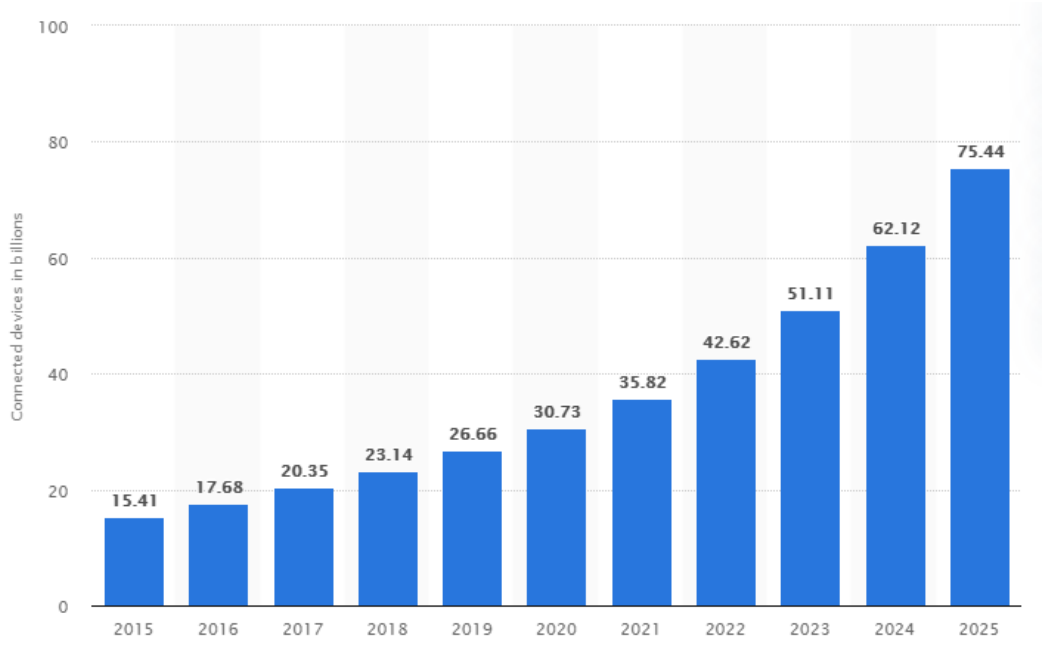


Figura 1-1 Número de dispositivos IoT conectados entre 2015 y 2025 (en miles de millones)

Fuente: [3]

1.1.2. La Problemática de la integración vertical

Este nuevo concepto del uso de dispositivos trajo consigo varias problemáticas que deben resolverse por el potencial de información que representan, debiéndose una de ellas al acelerado crecimiento del número de dispositivos afectando a las reservas de direcciones IPv4, por lo que será necesario una implementación y masificación del uso de IPv6 para soportar este crecimiento [4].

La problemática que se tratará en esta tesis es el uso de modelos de integración vertical de los servicios IoT, donde para su desarrollo es necesario contar con la arquitectura completa de configuración y despliegue de sensores, conexión, comunicación, procesamiento de los datos y la aplicación. Esto ocurre por la gran variedad de dispositivos, sensores, protocolos de comunicación y métodos de conexión con una red de datos para los dispositivos, lo que genera que cada uno de los servicios se encuentre muy optimizado para la función que realice, pero que se convierte en muy inflexible a integrarse con otros servicios o aplicaciones [5, p. 14]. Además, requiere que los desarrolladores de aplicaciones aprendan o desarrollen librerías que permitan la comunicación con los protocolos propios de IoT, así como las mejores prácticas de

implementación y un sistema de registro para mantener un inventario de los dispositivos y la data que recolectan. Lo que resulta en tiempos de desarrollo más largos y evita que desarrolladores con pocos conocimientos de esta arquitectura aprovechen todos los beneficios que brinda IoT.

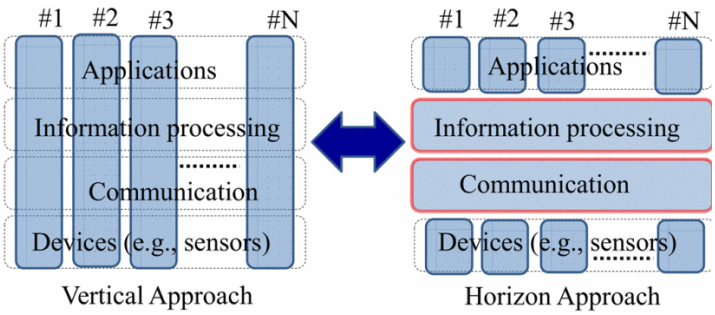


Figura 1-2 Integración vertical y horizontal en IoT

Fuente: [6, p. 1]

Por lo tanto, se está empezando a migrar al modelo de integración horizontal, donde se tiene un framework común para todos los servicios y se realiza la conexión de los dispositivos sin importar sus funciones, protocolos o formas de conexión y brindando una interfaz flexible y adaptable para la consulta y extracción de datos [7, p. 1]. Sin embargo, crear este modelo es un problema complejo porque requiere que cualquier dispositivo IoT sea capaz de conectarse a pesar de su heterogeneidad en cuanto a capacidad de procesamiento, tipo y número de sensores, protocolos de comunicación, uso de energía, etc. Otro aspecto importante a considerar es el alto número de dispositivos que pueden ser conectados al sistema y la separación necesaria que se requiere de acuerdo al usuario o aplicación a la que pertenezca cada uno de ellos. En consecuencia, se requiere de un sistema que sea compatible con todos los dispositivos IoT, escalable a demanda y con un sistema de autenticación y autorización para su uso por diferentes usuarios y/o aplicaciones.

1.1.3. Solución propuesta

Esta tesis propone el diseño de un *middleware* que brindará una capa de abstracción entre los dispositivos IoT y distintas aplicaciones y/o usuarios que requieran su información. Esto permitirá que los desarrolladores no necesiten conocer la implementación de las conexiones directas con estos dispositivos, sino contarán con

interfaces familiares que se usan en las aplicaciones web, como lo son las APIs Rest, que brinden toda la información generada por los sensores.

Para este fin, se usará un modelo de abstracción que englobe a todos los dispositivos IoT, el cual fue diseñado por el asesor en su tesis de posgrado y será ampliada e implementada para su funcionamiento en una arquitectura de microservicios. La forma como se soluciona la heterogeneidad de los dispositivos IoT es con una estructura de datos dinámica que contiene campos principales que son compartidos con todos los dispositivos, pero que podrá extenderse con campos particulares definidos por cada dispositivo. Esa información será usada para que los dispositivos se registren con la información de sus capacidades en cuanto a sensores, el tipo de data que recolectan y *metadata* adicional que se usará para crear un modelo virtual de cada uno de ellos. Luego, para solucionar el problema de los diferentes protocolos de comunicación IoT se implementarán conectores para cada uno de los protocolos IoT brindando la capa de conexión necesaria para su uso [5, pp. 40–45].

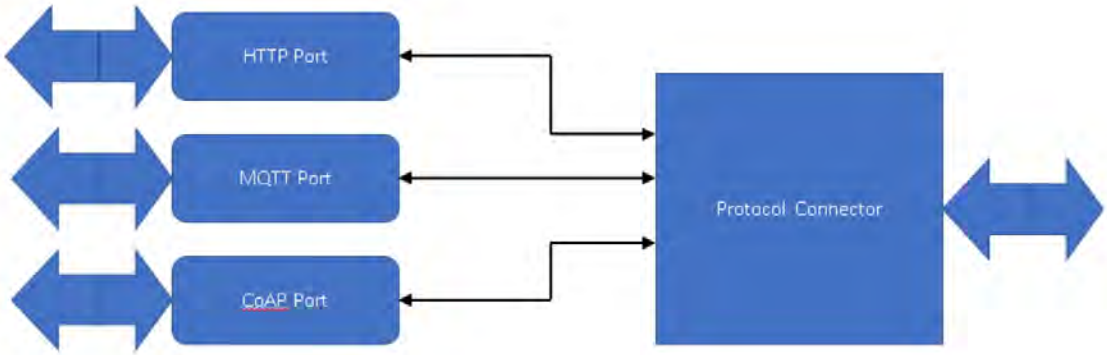


Figura 1-3 Capa de conexión del modelo diseñado por el asesor

Fuente: [5, p. 41]

El sistema también cuenta con una capa de almacenamiento de datos, el cual consistirá de una base de datos donde se guardará la información de cada uno de los dispositivos, su modelo virtual y las mediciones que realicen. Como se requiere que la caracterización de cada uno dispositivos se realice con una estructura de datos dinámica, se usará una base de datos no relacional porque ofrecen “un modelo de almacenamiento optimizado

para los requerimientos del tipo de dato que se está guardando" [8] y para almacenar los datos históricos recolectados se usará una base de datos de marcas temporales (*Time Series Database*) la cual está optimizada para guardar información que contenga marcas temporales y sus cambios en el tiempo [9].

Finalmente, todo este sistema será diseñado para que funcione en una arquitectura de microservicios, donde cada uno de sus componentes será un servicio que podrá ser implementado y desplegado independientemente, solamente exponiendo una interfaz de datos para interactuar con el usuario o con otros servicios. Esto permite que el desarrollo de nuevas funcionalidades sea más rápida y fácilmente integrable, por lo que cualquier desarrollador puede extender el sistema aún sin conocer completamente la estructura completa del sistema, sino solo los servicios con los que tiene que interactuar. Sin embargo, este sistema tiene como desventaja que, al ser distribuido, se presentan problemas como la inconsistencia de datos, la comunicación entre servicios y la implementación en sí se vuelve mucho más compleja [10].

1.2. Objetivos de la tesis

1.2.1. Objetivo general

Diseño e implementación de un middleware que permita la conexión y abstracción de los dispositivos IoT con diversas aplicaciones o servicios usando la arquitectura de microservicios.

1.2.2. Objetivos específicos

- Diseño e implementar de un sistema que permita a los dispositivos IoT de diferentes características y protocolos de comunicación enviar información de forma estandarizada usando modelos virtuales de los equipos.

- Diseño e implementación de un sistema de almacenamiento de la data recolectada y su consulta a través de interfaces API Rest para su consumo por usuarios y aplicaciones web.

- Diseño e integración de los sistemas anteriores en una arquitectura de microservicios permitiendo su alta disponibilidad y escalabilidad de acuerdo a la carga recibida.

- Ejecución de pruebas de desempeño con dispositivos IoT simulados para medición de escalabilidad y tiempo de respuesta; datos que serán comparados con otros *middlewares* IoT.

1.3. Razones que motivaron el desarrollo del trabajo

La presente tesis se desarrolla como respuesta a las dificultades que se observaron en la ejecución de proyectos IoT de parte del Grupo IoT – PUCP. Esto se debe a la barrera de entrada que supone el aprendizaje del funcionamiento de los diversos protocolos de comunicación IoT y sus implementaciones en los lenguajes de programación usados en los proyectos. Con la introducción del *middleware* propuesto, el tiempo que le toma a un nuevo integrante a usar la data de los dispositivos será reducido considerablemente y será destinado al procesamiento y análisis de la información recolectada. Además, servirá como base para futuros proyectos de investigación o desarrollo por su modularidad y extensibilidad.

1.4. Formulación del diseño del *middleware*

En base a los objetivos antes presentados, se formulará el diseño del *middleware* propuesto presentándose sus requerimientos y la selección de los componentes a usar.

1.4.1. Requerimientos

1.4.1.1. Requerimiento de abstracción de los dispositivos

Como se requiere que el sistema sea capaz de conectar con todos los dispositivos IoT actuales y futuros, sin que importe el proveedor, número de sensores o tipo de información que envíe, el sistema debe ser capaz de modelar y abstraer todos ellos a representaciones virtuales que sean tratados como una entidad que es una fuente de datos. Para ello, cada uno de los dispositivos se registrará en el sistema con un mensaje que indique las diferentes propiedades con las que cuenta como es el número de sensores, el tipo de medición que realizan y sus respectivas unidades. Además, cada uno

de estos dispositivos se encontrará asociado a un usuario del sistema mediante un "token de registro", el cual tendrá que especificarse como un campo más del mensaje de registro. Para permitir que esta información tan diversa sea procesada por el sistema, se necesita de una estructura de datos dinámica, una capa de almacenamiento capaz de procesarla y que pueda trabajarse con distintos lenguajes de programación.

1.4.1.2. Requerimiento del almacenamiento de los datos recolectados

Como los datos históricos que van a almacenarse cuentan intrínsecamente con una componente de tiempo, se requiere de una capa de almacenamiento que sea capaz de almacenar eficiente información teniendo como su base principal las marcas de tiempo. Además, cada una de las mediciones guardadas debe ser capaz de contener metadata que sea capaz de identificarla por el dispositivo, tipo de medición o unidad usada. Por otra parte, esta capa de almacenamiento debe ser capaz de consultar eficiente su registro de datos de forma puntual o en un periodo dado de tiempo.

1.4.1.3. Requerimiento de los conectores de protocolos en la capa de comunicación

Los dispositivos IoT usan diferentes protocolos para conectarse con los diversos servidores para el almacenamiento de sus datos recolectados. Estos protocolos están pensados para ser usados con procesadores poco potentes y donde el tamaño de los paquetes es pequeño [5, p. 24]. Por ello, se han creado nuevos protocolos de comunicación que sean más eficientes en el uso de recursos del dispositivo y mucho menos complejos que los protocolos usados por computadoras o celulares. Como se observa en la Tabla 1-1, se cuenta con un gran número de estos protocolos por lo que será necesario limitar este número a solo 3 de la capa de aplicación que cumplan con la restricción de que funcionen sobre la capa IPv4/IPv6. La selección de estos protocolos será por su cuota del uso y popularidad en la comunidad IoT. Sin embargo, la aplicación admitirá extender el número de protocolos soportados mientras cumplan con la restricción antes mencionada.

Tabla 1-1 Protocolos usados por los dispositivos IoT

Capa de aplicación	HTTP, XMPP, CoAP, MQTT, AMQP
Capa de comunicación	IPv4, IPv6, 6LoWPAN, RPL
Capa de acceso	IEEE 802.15.4, Wifi, Ethernet, GSM, CDMA, LTE

Fuente: Adaptado de [11]

1.4.1.4. Requerimiento de interfaz de usuarios y gestión de dispositivos

En cuanto a la interfaz que usarán los usuarios y gestión de dispositivos se requiere lo siguiente:

- Se encuentre disponible a través de una interfaz API Rest, para su facilidad de uso y conexión con otras aplicaciones web.
- Proporcione autenticación a través de usuario y contraseña o la creación de una cuenta en el sistema. Cada uno de estos usuarios deberá contar con un "token de registro de dispositivo" el cual será generado al momento de la creación de cuenta.
- Permita listar los dispositivos a y listar las características del modelo virtual generado como el número de sensores, tipo y unidad de medición.
- Proporcione un *endpoint* que será usado para las consultas de la data recolectada por el sensor de cada dispositivo. Los tipos de consulta disponible serán el último registrado o por un periodo.

1.4.1.5. Requerimiento de escalabilidad y extensibilidad

Como se mostró en la Figura 1-1, el número de dispositivos IoT va a aumentar considerablemente en los próximos años y con ello la cantidad de datos que se recolecten. Por ello, se requiere contar con una aplicación que sea capaz de escalar eficiente y automáticamente de acuerdo a la carga recibida sin que se produzca fallas o

caídas del sistema. Esto se logrará escalando solo aquellos componentes que lo requieran para minimizar el consumo de recursos de computación. Además, se requiere que esta aplicación sea completamente extensible mediante la fácil incorporación de nuevos módulos de acuerdo a las necesidades de cada usuario.

1.4.2. Selección de componentes

Tomando como base los requerimientos antes mencionados, se procederá a definir y argumentar la selección de cada uno de los componentes que serán usados en la implementación del *middleware*.

1.4.2.1. Arquitectura de microservicios

Como se analizó en [12, p. 3], los nuevos desarrollos de aplicaciones requieren ser diseñados con "capacidades que soporten accesos simultáneos y masivos de los usuarios, disponibilidad continua a lo largo del año, y con mejoras constantes que agreguen funcionalidades pedidas por los usuarios en cada actualización". Además, con la aparición de los modelos de computación en la nube, se necesitan de patrones de diseño de aplicaciones que aprovechen completamente esta nueva capacidad de cómputo y servicios que se brindan [12, p. 4].

La arquitectura de microservicios es un patrón de diseño que permite responder y solucionar a los problemas y necesidades antes mencionados. En ella, los módulos de la aplicación se estructuran como un conjunto de servicios distribuidos e independientes que cumplen una función específica. Esto permite que cada una de las aplicaciones se pueda diseñar e implementar de forma independiente, usando las mejores herramientas y lenguajes de programación dependiendo de los requerimientos que se necesiten en cada una de ellas, además de la experiencia y criterio del equipo de desarrollo [10].

Por ejemplo, se puede usar C en un microservicio que requiera tener el mayor rendimiento posible o Python si se necesita trabajar con librerías de análisis de datos. O se pueden usar sistemas de bases de datos especializadas según la función que realice

el servicio, como podría ser Elasticsearch enfocado principalmente en el búsqueda de la información de cualquier tipo, ya sea texto, números, datos geoespaciales, etc. [13].

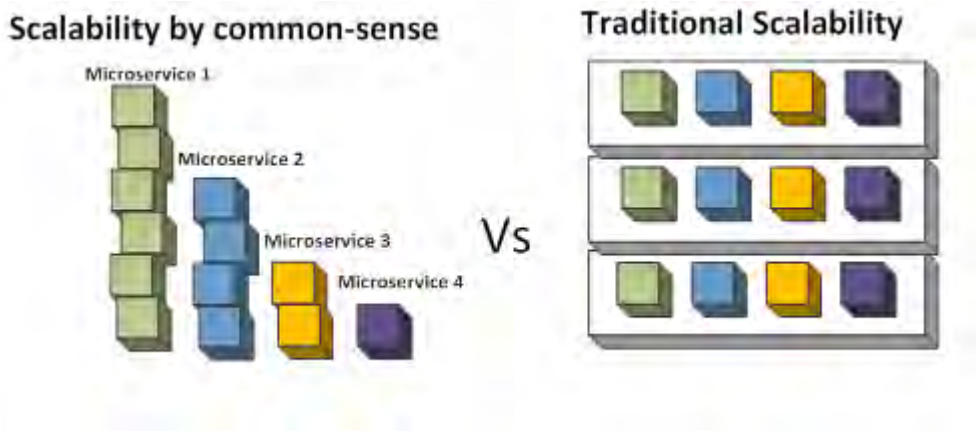


Figura 1-4 Diferencia en el escalado de microservicios y una aplicación monolítica

Fuente: [14]

Otra ventaja importante que otorga esta arquitectura, es la posibilidad de escalar cada uno de los servicios de forma independiente, asignando la mayor parte de la potencia computacional a aquellos que procesen la mayor cantidad de solicitudes, evitando que los servicios menos cargados malgasten estos recursos. Esta propiedad de los microservicios permite que se cumpla con el requerimiento de escalabilidad y también con el de extensibilidad, por lo que fue seleccionado por estos motivos.

Esta arquitectura también permite que cada uno de los equipos de trabajo pueda usar tecnologías *DevOps*, mejorando su "capacidad de responder mejor a las necesidades de los clientes, aumentar la confianza en las aplicaciones que crean y alcanzar los objetivos empresariales en menos tiempo" [15]. Dos métodos que se van a aplicar en la implementación del *middleware* es el de la Integración Continua y Despliegue Continuo, los cuales permiten que tanto los cambios que se realicen en los repositorios del código fuente de cada servicio, así como su despliegue estarán completamente automatizados para que las pruebas y extensión de funciones se realicen de la forma más eficiente y rápida posible [16].

Tabla 1-2 Definiciones de CI/CD en DevOps

Característica	***Continuous Integration* (CI)**	***Continuous Delivery* (CD)**	***Continuous Deployment* (CD)**
¿Qué hace?	Fusionar el código nuevo elaborado con la línea principal del código de la aplicación en el repositorio compartido.	Automatiza todo el proceso de publicación del software.	Integra segmentos de código en el entorno de producción para que los clientes puedan usarlos.
¿Cómo lo hace?	Verifica el código, compila, lo integra a la aplicación principal y realiza pruebas.	Compila, realiza pruebas adicionales y cargan el código al entorno de prueba	Verifica el código y lo envía al entorno de producción.
¿Qué soluciona?	Tener exceso de divisiones de la aplicación al mismo tiempo que podrían generar conflicto.	Tener poca visibilidad y comunicación entre los equipos comerciales y de desarrollo	El sobrecargar a los equipos de operaciones con procesos manuales que retrasan la distribución de las aplicaciones

Fuente: [16]–[18]

1.4.2.2. OpenAPI

Para el diseño de las interfaces Rest de los diversos servicios se cuentan con varias herramientas para su descripción. Entre ellas la que más destaca es OpenAPI, la cual es una especificación que permite declarar como código el funcionamiento de diversas interfaces para el intercambio de programación, donde se describen las rutas con las que se cuenta, los parámetros necesarios para una ejecución exitosa y las posibles respuestas que se entregan, facilitando la lógica de la implementación de los clientes. SwaggerHub es una plataforma que permite generar de forma sencilla estos documentos, además de servir como repositorio para que personas externas tengan esa declaración de la interfaz para usarla [19].

1.4.2.3. Tipo de comunicación entre los diversos microservicios

Con el trabajo de investigación realizado en [12], se mostraron dos tipos de sistemas de comunicación que pueden usar los microservicios para el intercambio de información, el síncrono usando el modelo Rest u otros similares, o el asíncrono mediante el uso de eventos. Se concluyó que para la comunicación entre los microservicios es mucho más conveniente usar el sistema asíncrono por diversas ventajas, mientras que para implementar una interfaz externa es mejor usar un sistema síncrono. Sin embargo, tener un sistema de comunicación completamente basado en eventos cuenta con una alta

complejidad adicional, aparte de contar con un diseño mucho más elaborado en cómo se organiza el manejo de los eventos dentro de cada microservicio. Por lo tanto, este sistema de comunicación será usado donde se requiera un alto tráfico de datos y sea simple de implementar, mientras que se usarán sistemas de comunicación basados en Rest donde la comunicación sea más compleja o no se requiera de una alta eficiencia. De la misma forma, se implementará una interfaz API Rest para la comunicación con los usuarios y aplicaciones externas, pero podrá extenderse a nuevos protocolos como gRPC o GraphQL gracias al uso de microservicios.

Para poder usar este sistema de comunicación es necesario contar con un *broker* que se encargue de envío de paquetes a los servicios interesados de acuerdo al tipo de evento que se produzca. Existen varias implementaciones de *brokers* de código libre entre las que se encuentran RabbitMQ, Apache Kafka, etc. de entre las cuales se usará RabbitMQ por su bajo consumo de recursos, alta escalabilidad y librerías para diversos lenguajes de comunicación [20].

De la misma forma, en los últimos años se cuenta cada vez más con soporte de funcionalidades asíncronas dentro de los mismos lenguajes de programación, lo que permite que, si algún comando se ejecuta por un lapso indeterminado de tiempo, como una petición web, se realicen otras actividades mientras se espera que regrese el resultado. Este paradigma de programación se encuentra presente en Python mediante la palabra *async* y *await*, y en Java mediante las librerías *Reactor* y *Webflux* [21], [22].

1.4.2.4. Api Gateway

Otro elemento importante dentro de esta arquitectura es el *Api Gateway* encargado de servir como puerta de entrada para las llamadas que se realicen desde fuera de la aplicación, como es el uso de la interfaz de usuario definida anteriormente. Además, va a poder realizar las llamadas correctas a los servicios necesarios de acuerdo a los recursos que se quieran acceder, traduciendo también protocolos o tipos de comunicación que no acepte el cliente conectado. Como se encuentra en el perímetro de la aplicación, es el elemento perfecto para implementar la seguridad de la aplicación,

ya que ninguna solicitud debe llegar a uno de los servicios si es que no se encuentra autenticada. Para este elemento se usará una implementación de *Api Gateway* como Ambassador o Netflix Zuul, pero también se verá la posibilidad de implementar uno si el seleccionado no cumple con los requerimientos pedidos.

1.4.2.5. Capa de almacenamiento de datos

Como se vio en los requerimientos de la aplicación, son necesarias dos bases de datos importantes para el su funcionamiento. Una es la que se encarga de guardar los modelos virtuales de los dispositivos que se registren en el sistema, la cual debe soportar estructuras de datos dinámicas y tenga disponibles librerías en varios lenguajes de programación. El tipo de bases de datos que cumplen con estas características son las llamadas no relacionales como MongoDB, el cual almacena la data en un formato flexible que puede ser cambiado constantemente, es altamente escalable y es de código abierto [23].

La segunda base de datos importante que se va a necesitar es la encargada de guardar el registro histórico de las mediciones que envíen los dispositivos IoT. Como esta contiene información temporal de forma intrínseca, se debe seleccionar una base de datos optimizada para guardar información en función del tiempo. Por lo tanto, se va a usar una base de datos de marcas temporales o TSDB porque están especializadas en trabajar con data que cuenta con información temporal, siendo InfluxDB la implementación a usar por su popularidad y librerías para diferentes lenguajes de programación.

	MongoDB	**InfluxDB**
Ventajas	Base de datos orientada a documentos y no relacional.	Base de datos de series de tiempo, diseñados para manejar datos relacionados con el tiempo.
	Fácil de escalar.	Eficiencia en la utilización de los recursos.
	Configuración e instalación simple.	Se puede ejecutar una consulta y escribir los resultados directamente en la base de datos con su característica *into clause*
	Sin esquema	Todos los datos se fragmentan para permitir el paralelismo y el escalado horizontal.
	Alto rendimiento.	Excelente rendimiento

	Usa memoria interna para almacenamiento lo que permite un acceso más rápido a los datos.	Se puede definir cuanto tiempo se desea conservar los datos con su Política de retención
Desventajas	No hay soporte para realizar transacciones.	El rendimiento puede disminuir si no se diseña correctamente
	Carece de *Joins.* Para consultar datos de dos o más colecciones se debe hacer más de una consulta.	Se pierde todo el fragmento de datos cuando la fecha de retención caduca debido a la limitación de *backup* de su Política de retención

Fuente: [24], [25]

1.4.2.6. Autenticación y autorización

OAuth2 es un protocolo que permite a los usuarios acceder a los recursos protegidos en cada uno de los servicios con mínimo tráfico para la comprobación de la identidad. Esto se logra mediante tokens de acceso, los cuales cuentan con la información necesaria para que un servicio pueda asegurarse de que el usuario que desea acceder se encuentra autenticado y autorizado por una entidad en la que confía. Para ello se requiere que el usuario inicie sesión en un proveedor de identidades, lo que le otorga un JSON Web Token (JWT) el cual contiene la información firmada de la identidad del usuario y sus capacidades, con la cual puede realizar consultas en diferentes servicios asociados a ese sistema [26].

1.4.2.7. Docker

Docker es una herramienta que permite a las aplicaciones empaquetarse con todas sus librerías y dependencias para que puedan ser desplegadas de forma rápida y sencilla en diferentes entornos de computación. Son consideradas métodos de virtualización, pero a diferencia de máquinas virtuales, no cuentan con el costo adicional de todo un sistema operativo, si no que usan el *kernel* del *host* donde se encuentran desplegadas [27].

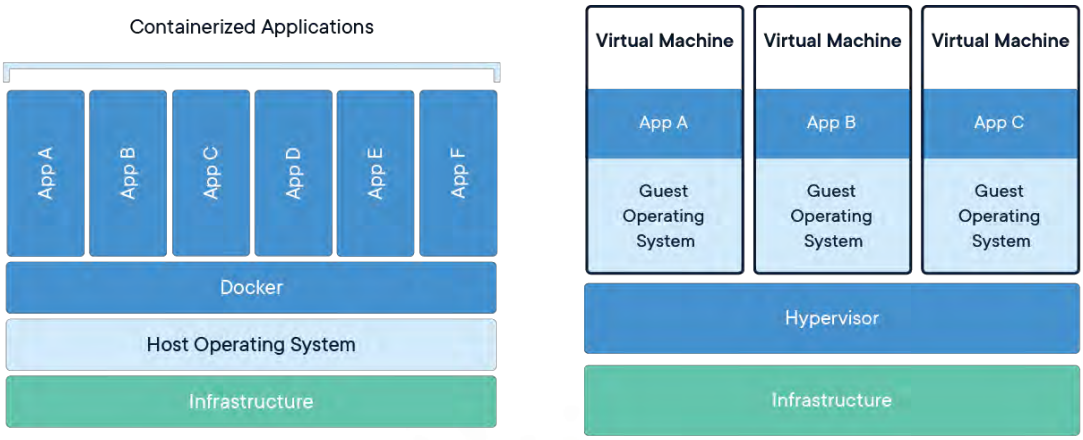


Figura 1-5 Diferencia entre contendores y máquinas virtuales

Fuente: [27]

Funciona por medio de imágenes y contenedores, donde una imagen representa un *snapshot* del estado de la aplicación y sus dependencias. Cuando se necesita ejecutar una copia de la aplicación, se usa su imagen como base para crear un contenedor, el cual podrá interactuar con el sistema operativo u otros contendores que se encuentren en ejecución [28].

Es una pieza fundamental de la implementación de la arquitectura de microservicios porque se encarga de la escalabilidad del sistema. Si un servicio necesita ser escalado, entonces solo es necesario ejecutar un contenedor más usando como base su imagen. Además, su portabilidad permite que sea ejecutado en cualquier proveedor de nube, los cuales incluso brindan servicios especializados para trabajar con imágenes y contenedores Docker.

1.4.2.8. Kubernetes

Kubernetes es una plataforma que permite que se encarga de administrar aplicaciones empaquetadas en contenedores, encargándose de orquestar la infraestructura necesaria para que puedan ejecutarse correctamente. Esto lo logra mediante una configuración declarativa en la que se define el estado deseado de la aplicación, y esta herramienta se va a encargar de alcanzar y mantener dicho estado. Ahí se definen las características con las que debe de contar cada uno de los componentes de la aplicación, como el número de contenedores que deben ejecutarse para uno de ellos, la capacidad

máxima de escalado, el uso de recursos de los sistemas y muchos otros más. Es capaz de funcionar en un modelo de clúster, donde se cuenta con varias máquinas virtuales o físicas controladas por un nodo principal encargando de distribuir los contenedores de acuerdo a los requerimientos antes mencionados [29].

Aunque cuenta con un amplio número de elementos, los conceptos de *pods* y *services* ayudan a comprender una característica importante que brinda. El primero es un envoltorio que se le asigna al contenedor de una aplicación o servicio y que otorga métricas a Kubernetes que serán usadas para mantener el estado declarado de la aplicación. Estas son definidas y ejecutadas en la aplicación de acuerdo a la configuración dada, pero necesitan de un elemento para comunicarse con el mundo exterior, el cual será otorgado por un *service*, el cual se encarga de exponer una sola dirección IP para acceder a cualquiera de las réplicas y balancea su carga de forma equitativa [30].

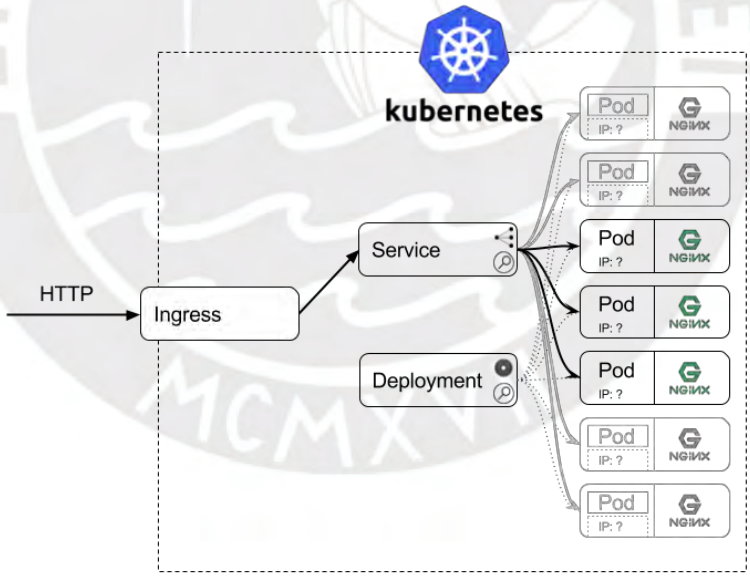


Figura 1-6 Ejemplo de una aplicación desplegada en Kubernetes usando contenedores nginx

Fuente: [30]

Al igual que Docker, esta plataforma es perfecta para desplegarla en un servicio de computación en la nube por sus características de escalamiento y capacidad de crear y desplegar topologías de cómputo complejas a demanda. Además, los proveedores cloud ofrecen esta plataforma como un servicio, donde ellos se encargan de la infraestructura

del clúster y su mantenimiento, y el desarrollador solo brinda la configuración final del despliegue de su aplicación.

1.4.2.9. Servicio de descubrimiento de servicios

Como se observó en la sección anterior, Kubernetes se encarga de la el despliegue y orquestación de los microservicios que componen una aplicación, manejando las réplicas y direcciones IP con las cuales acceder a ellas. Sin embargo, estas direcciones con las que se cuentan son dinámicas, porque los *services* que se crean pueden realizarse en cualquier nodo del clúster y en momentos indeterminados. Por lo tanto, es necesario contar con un servicio que permita mapear los servicios con una dirección a la que puedan comunicarse. Por ejemplo, el servicio encargado de registrar dispositivos requiere conectarse con la base de datos para almacenar esta información o con el *broker* de mensajes para que envíe la información de un registro exitoso [31].

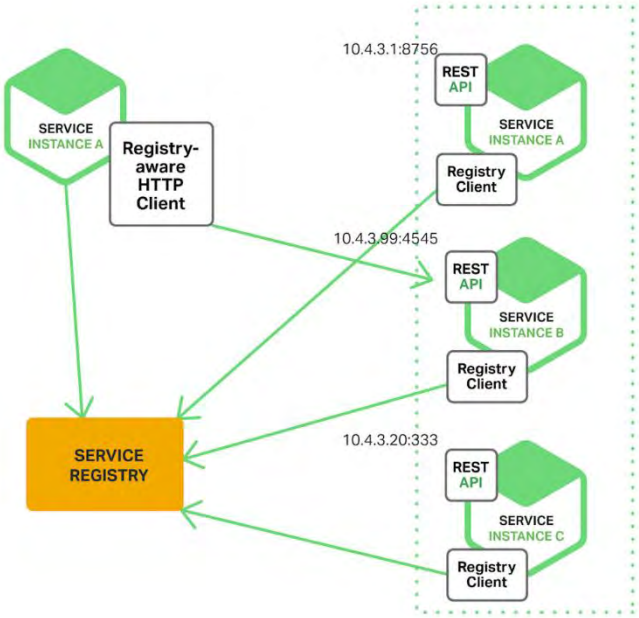


Figura 1-7 Funcionamiento de un servicio de descubrimiento de servicios

Fuente: [31]

Kubernetes ofrece una solución de descubrimiento de servicios incluida en su propia infraestructura, la cual está basada en el protocolo DNS, proveyendo a cada uno de los servicios de un FQDN que puede ser usado por cualquier otro servicio dentro del

despliegue [32]. También existen otros softwares que realizan esta misma función como Consul, pero que agregan funcionalidades como conexión segura mediante proxies, comprobación del estado de los servicios, o funcionamiento con otras infraestructuras fuera de Kubernetes [33]. Para este desarrollo, como toda la aplicación será desplegada en Kubernetes, se usará el servicio que se encuentra integrado en él, pero también se investigará el uso y despliegue de Consul como parte de la experimentación.

1.4.2.10. Lenguajes de programación y *frameworks*

Teniendo la elección de los componentes de la infraestructura, se deben seleccionar los lenguajes de programación que se usarán para el desarrollo de cada uno de los servicios. Estos deben contar con un *framework* que permita implementar un servidor para exponer su interfaz, desarrollar la lógica necesaria y conexión con otros servicios a través de librerías como las de MongoDB, InfluxDB, RabbitMQ, etc.

Entre los lenguajes que más sobresalen con estas características son Java, Python y Golang; los cuales cuentan con los *frameworks* Spring Boot Cloud, Flask y Gomicro respectivamente.

Tabla 1-3 Comparación entre Golang, Java y Python

	Golang	**Java**	**Python**
Ventajas	Proporciona gran velocidad y soporte para la concurrencia.	Muchos servicios en la nube escalan con este lenguaje.	Código legible
	Proporciona potentes bibliotecas estándar para crear servicios web.	Proporciona muchos recursos de programación y bibliotecas.	Facilidad para integrarse con otras tecnologías
	Permite construir aplicaciones grandes y complejas.	Facilita el experimentar con otros lenguajes o marcos sin hacer una gran inversión.	Alta productividad en todo el ciclo de vida del desarrollo.
Desventajas	La seguridad solo se ejecuta durante el tiempo de compilación	Bajo rendimiento	Ejecución lenta
	Sin gestión manual de memoria	Sin soporte para programación de bajo nivel	Alto consumo de memoria y muchos límites de diseño

Fuente: [34]

Tabla 1-4 Comparación entre los frameworks de microservicios

	Spring Boot	**Flask**	**Go-micro**
Soporte de microservicios	Spring Cloud contribuye con descubrimiento de servicios, balance de carga, corte de circuito, rastreo distribuido y monitoreo	Mediante el uso de la librería Flask-Restful permite crear microservicios Rest de forma rápida.	Micro contribuye con un conjunto de librerías herramientas orientadas al desarrollo principalmente en el lenguaje Go.
Librerías de mensajería de eventos con RabbitMQ	Spring AMQP's RabbitTemplate	Pika	Streadway AMQP
Librerías para conexión con MongoDB	MongoDB driver	Pymongo	Mongo-go-driver
Librerías para conexión con InfluxDB	InfluxDB-client-java	InfluxDB-client	InfluxDB-client-go

Fuente: [35]–[37]

Con las diferencias descritas anteriormente se seleccionará el mejor *framework* para el tipo de servicio al momento del diseño. Sin embargo, la naturaleza de los microservicios permite que nuevos servicios o módulos usen las herramientas preferidas del desarrollador, por lo que podrá usar el lenguaje de programación o *framework* que mejor realice el trabajo o se le facilite.

1.4.2.11. Protocolos de aplicación IoT

En cuanto a los protocolos IoT, serán seleccionado aquellos más usados dentro del Grupo IoT – PUCP, entre los que destacaron 2 principales: MQTT y AMQP.

Tabla 1-5 Comparación entre AMQP y MQTT

	AMQP	**MQTT**
Uso de broker	Si	Si
Patrón de mensajería	*Publish-Subscribe*	*Publish-Subscribe*
Actualizaciones del servidor	*Push*	*Push*
Soporte de *WebSockets*	Sí	No
Librerías compactas (<1MB)	No	Sí
Binary payloads	Sí	Sí

Fuente: [38]

Para los diferentes protocolos de conexión, se escogen los diferentes *brokers* que se usarán, agregando también el protocolo HTTP.

- HTTP: aunque no sea un protocolo hecho específicamente para dispositivos IoT, este sigue siendo usado para conexiones con placas de mayor capacidad de procesamiento. Para este caso, se usará el mismo servidor web que se usa para la interfaz de usuarios y gestión de dispositivos.

- MQTT: se usará el *broker* Mosquitto de Eclipse que es Open Source. Implementa las versiones de MQTT 5.0, 3.1.1 y 3.1, además de ser ligero y tener compatibilidad con diferentes arquitecturas de procesadores [39].

- AMQP: se usará el broker RabbitMQ, el cual contiene librerías para interactuar con diferentes lenguajes de programación. Implementa las versiones de AMQP 0.9, 0.9.1 y 1.0 [40].

- Otros protocolos: al ser cada conector un servicio independiente, se puede extender a otros protocolos implementando el nuevo broker y conectándolo al sistema mediante las interfaces que se van a implementar.

1.4.2.12. Estructuras de datos a utilizar

Para modelar los dispositivos se usará el formato JSON, el cual es ligero, simple de interpretar y generar. Es el formato usado por preferencia en las peticiones web, lo cual lo ha vuelto popular e implementado en varios lenguajes de programación. Al no contar con un esquema fijo, es posible crear estructuras dinámicas que puedan representar de forma más exacta el dispositivo y permite la comunicación sencilla entre servicios [41].

Capítulo 2. Estudio de *middlewares* IoT alternativos

En este capítulo se estudiarán diversos *middlewares* alternativos que surgieron por los problemas mencionados en el capítulo anterior, además de analizar otro tipo de soluciones que han surgido en la literatura respecto a la integración de dispositivos IoT y el desarrollo en una arquitectura de microservicios.

2.1. Estándares propuestos

Antes de poder revisar implementaciones de *middlewares* es necesario estudiar estándares que atacan el problema de la poca integración horizontal que existe en los entornos IoT.

2.1.1. oneM2M

Este estándar pretende cubrir la brecha existente de la falta de una capa de comunicación entre los diferentes protocolos en las comunicaciones *Machine to Machine* (M2M). Esto se quiere lograr sin modificar los estándares actuales, si no crear requerimientos, arquitecturas, especificaciones de API y soluciones de seguridad que

permitan la interoperabilidad de las redes IoT. Fue formada por las organizaciones de desarrollo de estándares del mundo, como son ARIB (Japón), ATIS (USA), CCSA (China), ETSI (Europa), TIA (USA), TSDSI (India), TTA (Corea del Sur) y TTC (Japón) [42].

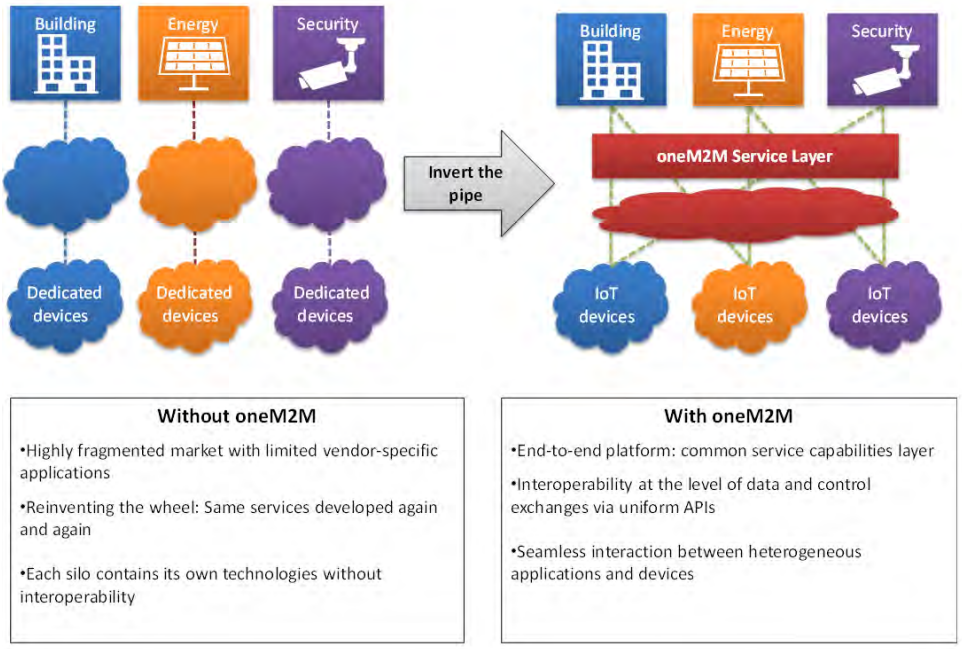


Figura 2-1 Capa de servicio propuesta por oneM2M

Fuente: [43]

Para lograr la integración horizontal se hacen uso de *Gateways* M2M que se encargan de conectarse con los dispositivos IoT sin importar el protocolo o tipo de comunicación que use, ya sea de capa de aplicación (AMQP, MQTT) o de capa de red (Zigbee, Bluetooth, RFID). Este *Gateway* se encarga de enviar toda la información recolectada a un servidor que cuente con interfaces con las cuales los usuarios u otras aplicaciones puedan acceder [44, pp. 13–18].

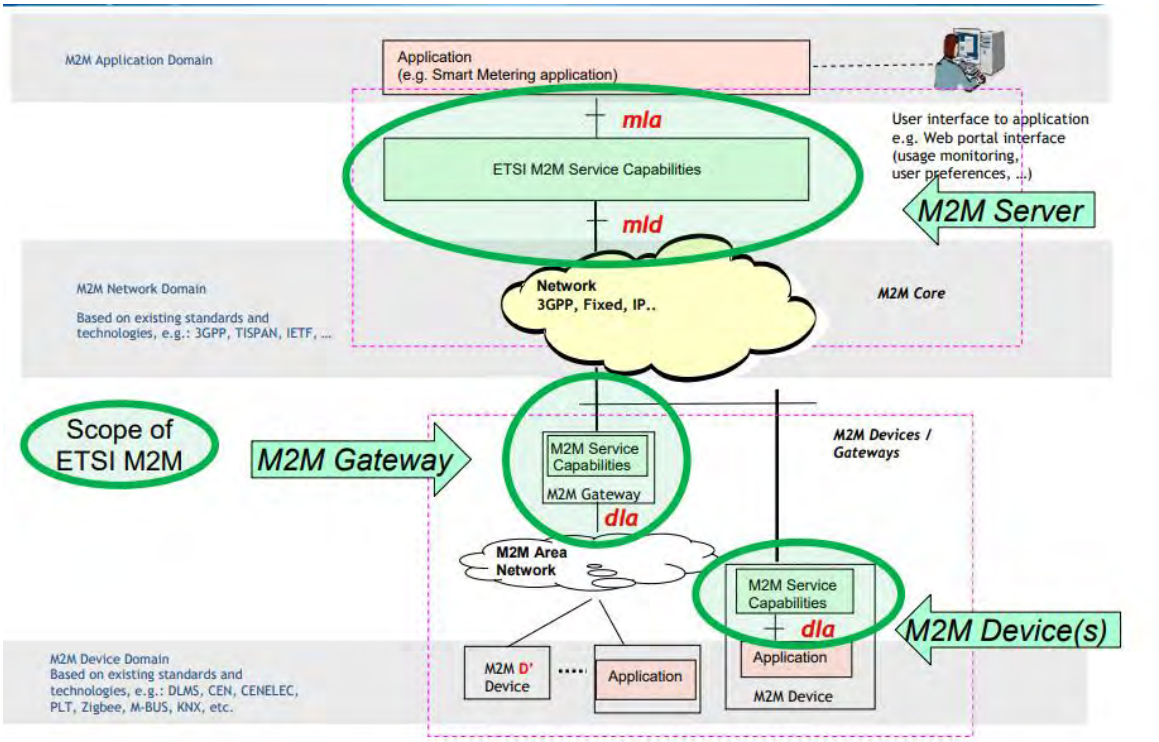


Figura 2-2 Resumen del sistema M2M a alto nivel

Fuente: [44, p. 16]

Esta especificación también propone más capacidades que los formulados en el *middleware* presentado en este documento, siendo una de ellas la comunicación punto a punto entre dispositivos, algo cada vez más relevante en el IoT Industrial donde diversos controladores usan la información que miden diversos sensores instalados en la etapa del proceso industrial para tomar decisiones ya programadas, pero que dependen de las circunstancias del entorno. Otro uso que va tomando relevancia y que necesita de este tipo de comunicación es el de los autos autónomos, donde cada uno de los vehículos toma decisiones de movimiento de acuerdo a sensores con los que cuenta o sensores que se encuentran en otros vehículos. Otras capacidades son el descubrimiento de dispositivos, seguridad y administración de estos equipos [45, p. 5].

Tabla 2-1 Beneficios empresariales que brinda oneM2M

Beneficios	
Menores costos CAPEX	Menor costo de despliegue Programadores se concentran en el desarrollo de las aplicaciones Contar con una capa de servicio horizontal permite ser reusada por varias aplicaciones
Menores costos OPEX	Comunicación eficiente (Por políticas o eventos) Se comparte la información del sensor (se produce una vez y se consume muchas veces) Se economiza en la capa de transporte (se usa la mejor red de transporte para las necesidades del negocio)
Reduce la fragmentación	Una capa de servicios común para diferentes segmentos reduce la necesidad de plataformas específicas para cada aplicación
Permite nuevas oportunidades de negocio	Se utilizan más recursos para la innovación en la capa de servicios y la compartición de datos permite innovación en las aplicaciones

Fuente: [46]

El *middleware* que se describe en este documento comparte la arquitectura de alto nivel de contar con conectores de distintos tipos de aplicaciones y una capa de servicio totalmente extensible. Por tanto, un trabajo a futuro sería el estudio y diseño de nuevos servicios que permitan cumplir los requerimientos que define esta especificación.

2.1.2. Especificación del Open Connect Foundation (OCF)

La Open Connect Foundation es una entidad que nació de la fusión de varias organizaciones como AllSeen Alliance, UPnP y Open Interconnect Consortium. Definieron una especificación que se encarga de diseñar una arquitectura centrada en la interoperabilidad, permitiendo a los desarrolladores lidiar con la complejidad de los despliegues IoT brindando un modelo común de datos que les permite interactuar con cualquier de estos dispositivos y la información que recolectan [47, p. 4].

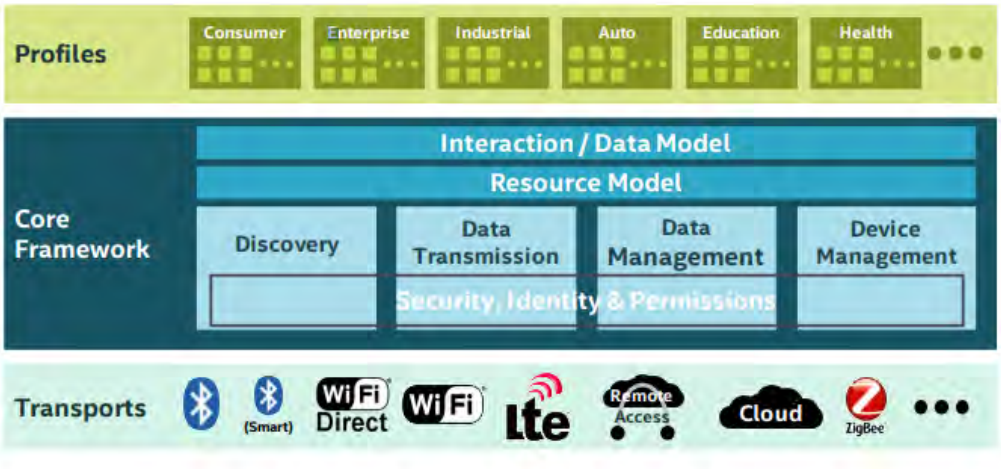


Figura 2-3 Especificación que usa el framework OCF

Fuente: [47, p. 6]

Como se observa en la Figura 2-3, la forma cómo implementan la arquitectura de esta especificación es mediante el uso de un *framework* que sirve como capa común entre la capa de transporte de los dispositivos IoT y diversas aplicaciones. Además. se observa que ese elemento cuenta con módulos que se encargan del descubrimiento, el manejo de la data y la administración de los dispositivos los cuales cuenta con seguridad por medio de la autenticación y autorización.

Esta especificación también presenta un modelo de abstracción de dispositivos basados en propiedades comunes y específicas a las cuales se acceden mediante URIs y usan el formato JSON y la especificación Swagger para el diseño de la interfaz que presenta cada uno de estos dispositivos [47, pp. 7–8], [48]. Se cuenta con el ejemplo del modelado de una impresora 3D en [49]

Figura 2-4 Ejemplo del modelado de un foco y ventilador por OCF

Fuente: [47, p. 8]

Existe un framework parte de OCF y predecesor a la especificación antes mencionada llamada AllJoyn, el cual permite la interoperabilidad y la comunicación M2M entre los dispositivos conectados. Esto se logra mediante un servicio de descubrimiento que reconoce a los dispositivos cercanos y crea una red dinámica con todos ellos. En este sistema se encuentra un AllJoyn Router encargado del envío de los mensajes a todos los dispositivos interesados, y se logra mediante un AllJoyn Bus, donde están conectados todos los dispositivos de un host [50].

Como principales desventajas se encuentra que el sistema solo trabaja sobre una conexión de área local (LAN), por lo que la extensión a dispositivos que se encuentren lejanos es complicada y otro es que de por sí no cuenta con un sistema de almacenamiento incluido, por lo que es necesario agregarle una extensión de almacenamiento de datos como AllJoyn Lambda [51]. Sin embargo, este framework no cuenta con los componentes necesarios para ser considerado un *middleware* de integración horizontal.

2.2. Implementaciones de *middlewares* IoT de integración vertical

En esta sección se presentarán dos implementaciones de las especificaciones anteriores, viendo su arquitectura y las características que presentan.

2.2.1. OM2M

Es una plataforma de servicios escrito en Java que provee las herramientas necesarias para la interoperabilidad de comunicaciones M2M (*Machine to Machine)*, basado en el estándar oneM2M y desarrollado por Eclipse. Desarrollado con una arquitectura modular, es posible extender su funcionalidad con plugins, siendo uno de ellos los proxies de comunicación para diferentes tipos de protocolos. La forma como funciona es que provee de una interfaz donde se crean y maneja los recursos M2M y existen procesos que se encargan de la autenticación, descubrimiento de servicios y más [52].

Esta plataforma provee un API RESTful para el intercambio de información usando XML, una arquitectura totalmente modular por medio del sistema OSGi, el cual permite añadir

y quitar plugins de la aplicación en tiempo de ejecución. Esta arquitectura se observa en la Figura 2-5, donde se cuenta con un procesador *CORE* encargado de procesar las solicitudes Rest de los clientes, procesador de protocolos de aplicación específicos como CoAP o HTTP, proxies de protocolos de trasporte como Zigbee y seguridad mediante el protocolo TLS-PSK. Cada uno de estos bloques está configurado como un plugin y puede añadirse soportes para diversos protocolos como MQTT, AMQP, LoraWAN, etc. [53, pp. 1081–1082].

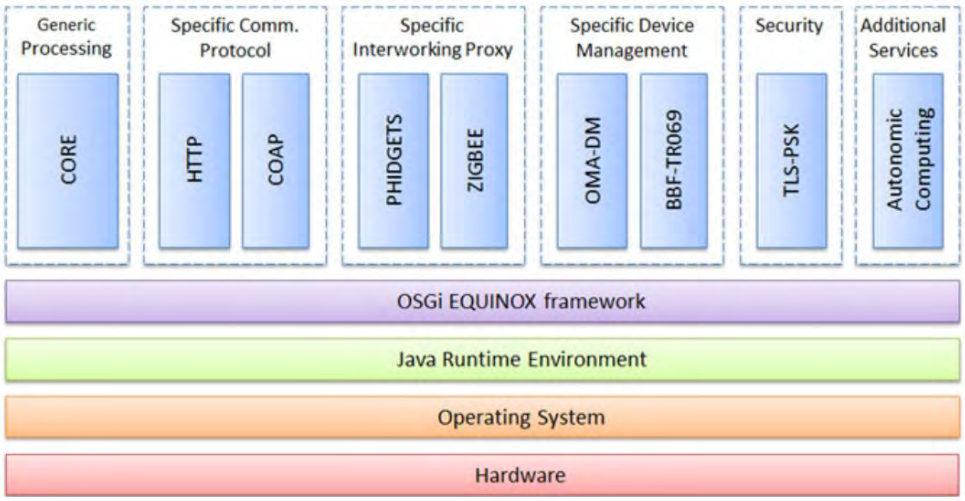


Figura 2-5 Arquitectura de OM2M

Fuente: [53, p. 1081]

Como esta aplicación es *open source*, se desplegará en el entorno de pruebas de obteniendo mediciones que sirven de base para la comparación del desempeño con el *middleware* presentado en esta tesis, siendo la métrica más importante el tiempo de respuesta y *overhead* que otorga.

2.2.2. IoTivity

Este proyecto es una implementación de referencia de la especificación OCF, mostrando las mejores prácticas que deben seguirse para desarrollar correctamente este tipo de *middlewares* [54]. Revisando su repositorio de código y la documentación con la que cuenta, aún se encuentra en un proceso temprano de desarrollo por lo que no será usado para la comparación como el middleware anterior, pero es importante resaltar su

existencia para que a futuro se analice si puede ser usado como un proyecto base de implementaciones siguiendo la especificación OCF.

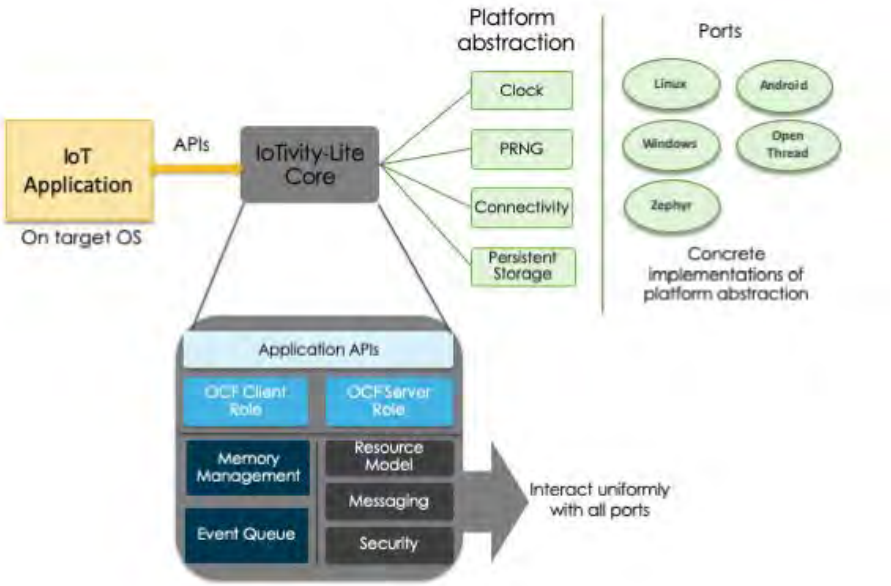


Figura 2-6 Arquitectura de funcionamiento de IoTivity

Fuente: [54]

Capítulo 3. Diseño e implementación del *middleware* usando la arquitectura de microservicios

En este capítulo se presentará el diseño y partes de la implementación del *middleware* usando la arquitectura de microservicios, presentando pruebas que se realizaron a cada una de los servicios por separado, integrándolos en un entorno local y finalmente su despliegue usando Docker-Compose.

3.1. Diseño lógico de la aplicación

En primer lugar, se va a presentar un diagrama del diseño lógico de la aplicación, mostrando la estructura final de los servicios, los canales de comunicación que existen entre ellos, la estructura de los mensajes que envían y los diagramas de flujo que siguen los dispositivos IoT. La especificación de cada uno de los servicios, como las interfaces que presentan y la descripción de su funcionalidad serán mostradas en las siguientes secciones.

3.1.1. Diagrama de diseño

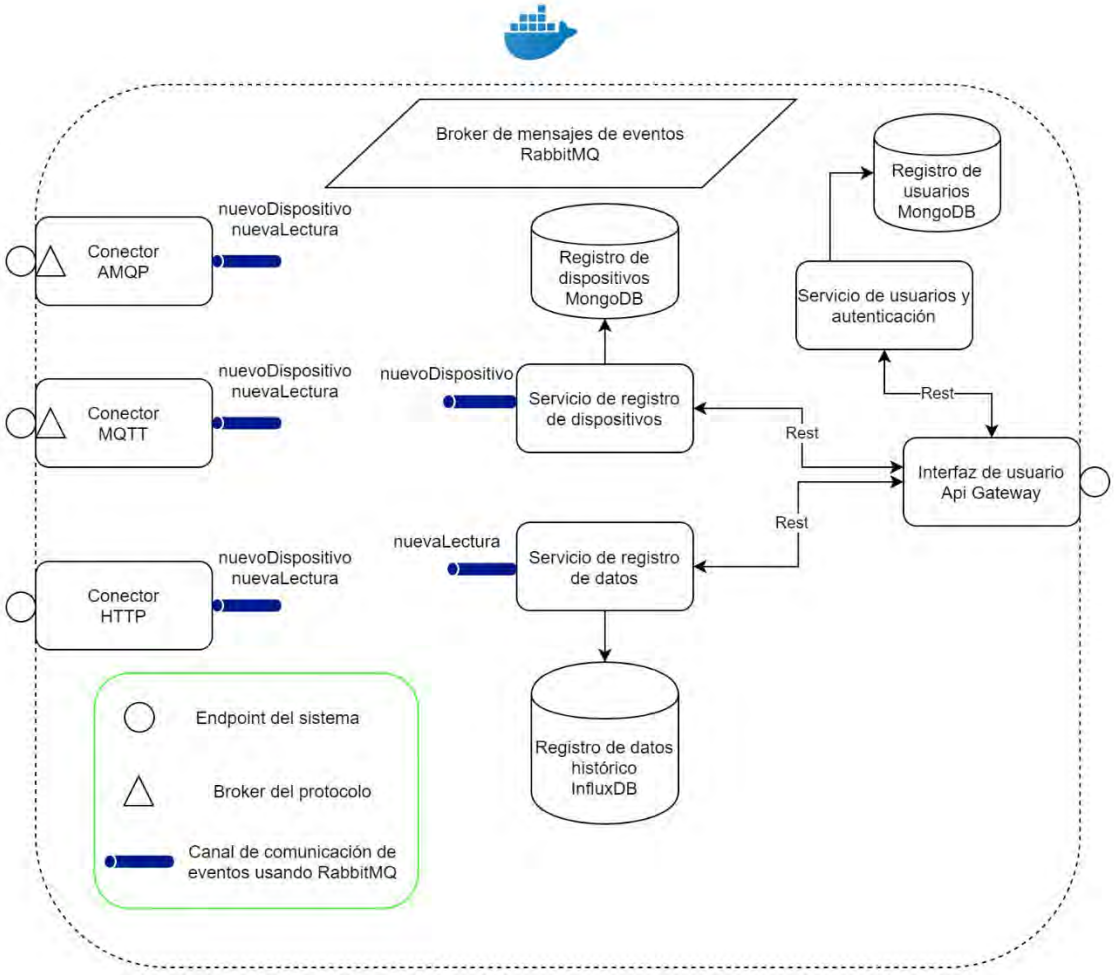


Figura 3-1 Diagrama lógico de la aplicación

Fuente: Elaboración propia

Como se observa en la Figura 3-1, se decidió usar un sistema de conexión asíncrona basada en mensajes de eventos entre los conectores de los protocolos IoT y los servicios que se encargan de registrar y guardar la información de los dispositivos. Esto se debe a la gran cantidad de información que va a cursar entre esos servicios, por lo que usar un protocolo síncrono como HTTP significaría contar con varias TCP que se crearan para cada nueva solicitud de registro o dato que llegue a la aplicación.

Por otro lado, en el lado de las comunicaciones entre los servicios de usuario se va a contar con una comunicación por medio de interfaces API Rest por el menor tráfico de datos que circula en esta parte de la aplicación. Aunque este tipo de comunicación sea

síncrona, la pérdida de eficiencia no será tan alta por el uso de programación asíncrona que se mostró en el primer capítulo.

Otro punto importante a resaltar es que los conectores y el *Api Gateway* son los únicos servicios donde se cuenta con comunicación externa, no se puede acceder de forma directa a los servicios de registro y de usuarios, ni a las bases de datos. Esto permite que cada uno de estos servicios pueda contar con interfaces que se usan para la comunicación entre los servicios de forma privada y otras que pueden exponerse para que los usuarios interactúen con ellas. Aparte, el servicio de usuarios y autenticación cumple dos roles, el de gestionar la información de los usuarios y de autenticar los tokens que los usuarios obtienen para acceder a su información almacenada en los demás servicios.

3.1.2. Estructura de los mensajes

Contar con una estructura de definida de los mensajes que reciben y envían entre los servicios permite que la comunicación se realice sin errores, ya que se detectan los mensajes mal formados o con datos incorrectos antes de que se realice alguna acción sobre ellos. Además, permite a los usuarios y futuros desarrolladores conocer los tipos de mensajes que requiere cada uno de los flujos presentes en la aplicación cumpliendo así el rol de documentar el uso y funcionamiento del sistema.

Como se seleccionó el formato JSON para el intercambio de mensajes, se utilizará la herramienta JSON Schema, la que permite que cada uno de los campos dentro de un mensaje JSON cuenten con una descripción de su uso, el tipo de dato que requieren, ejemplos, restricciones y más. Con esto, una aplicación puede comprobar si el mensaje que procesa cumple con las características descritas y enviar los mensajes de error de acuerdo a las validaciones que no se cumplen. Además, existen herramientas que permiten mostrar cada uno de estos esquemas de forma visual para que el usuario de la aplicación pueda formar correctamente los mensajes que envíen sus dispositivos a los servicios que lo requieran [55], [56].

En el caso del *middleware* a implementar, dos de los esquemas más importantes son los del registro de dispositivo y envío de datos, ya que ambos son parte de los flujos de datos que se presentarán en la siguiente sección y se requiere conocerlos para entender de mejor manera estos flujos. Sin embargo, los esquemas para cada uno de los tipos de datos se encontrarán en los documentos OpenAPI de los diversos servicios.

En primer lugar, los mensajes de registro cuentan con 4 campos obligatorios, los cuales son: "*operation*", "*ownerToken*", "*deviceID*" y "*virtualModel*" y uno opcional llamado "*additionalInfo*"

- El campo operación indica el tipo de transacción que va a ocurrir, ya sea un registro, actualización o hasta anulación del registro; sin embargo, el sistema solo permite el registro de un dispositivo en esta versión.

- El campo de token de usuario contiene un token que se genera para que se identifique a los dispositivos de un usuario en el servicio de usuarios. Cada uno de los usuarios puede contar con más de un token, lo que permite separar un grupo de dispositivos del mismo usuario.

- El ID del dispositivo contiene un identificador asignado por el usuario para reconocerlo entre dispositivos del mismo tipo y cuenta con la restricción de ser texto de 3 palabras separas por el carácter "/" y puede tener el formato de "*network*/*group*/*device*" para tener una jerarquía a la que pertenece cada uno de los dispositivos.

- El modelo virtual es una lista de las características con las que cuenta un dispositivo, donde cada uno de los elementos representa a un sensor o actuadores. Un sensor cuenta con cuatro campos, "*type*" que indica si es un sensor mediante la palabra "*reading*", "*readingType*" que indica el tipo de lectura que se realiza como temperatura, "*units*" que indica las unidades de las lecturas y "*dataType*" que indica el tipo de dato de la lectura. En el caso de los actuadores, como no se van a implementar en esta versión del sistema, solo

cuentan con un campo que los identifica como actuadores, pero no será usado por ningún servicio.

- El campo de información adicional puede usarse para enviar información adicional del dispositivo que quiere ser almacenada en la capa de almacenamiento de dispositivos.

Figura 3-2 Ejemplo de mensaje de registro

Fuente: Elaboración propia

Por otro lado, el esquema de los mensajes de nuevo dato cuenta con 3 campos obligatorios: "*objID*", "*ownerToken*" y "*readings*".

- El campo de ID de objeto cuenta con el identificador que entrega el servicio de registro de dispositivos luego del registro exitoso de un dispositivo, siendo diferente al ID de dispositivo que sirve más como un identificador jerárquico para los usuarios.

- El campo de token de usuario es el mismo que en el esquema anterior.

- El campo de lecturas es un arreglo donde se envía la lectura para uno o más sensores. Cada una de las lecturas cuenta con dos campos: "*index*" el cual indica la posición de la descripción del sensor en el modelo virtual del esquema anterior y "*reading*" donde se envía el valor de la medición en sí. Por ejemplo, si en el campo "*index*" se envía el valor 0, se usará la descripción del primer elemento de modelo virtual guardado enviado anteriormente.

Figura 3-3 Ejemplo de mensaje de envío de datos

Fuente: Elaboración propia

La estructura de los demás mensajes no será explicada tan a fondo, pero cuentan con un nombre los suficientemente auto descriptivo y comentarios adicionales en su esquema para que sean entendidos completamente.

3.1.3. Diagramas de flujo

En la aplicación se encuentran dos diagramas de flujo importantes, el de registro de un dispositivo y el de envío de datos. Existen más flujos en el sistema, pero no implican la conexión con más de dos servicios ya que solo son de consulta, por lo que serán explicados en la sección de cada dispositivo. De igual forma, las descripciones que se presentan en el diagrama son de alto nivel, por lo que el detalle del funcionamiento integral de cada uno de los recuadros también será explicado en la sección de cada uno de los servicios que se presentará más adelante.

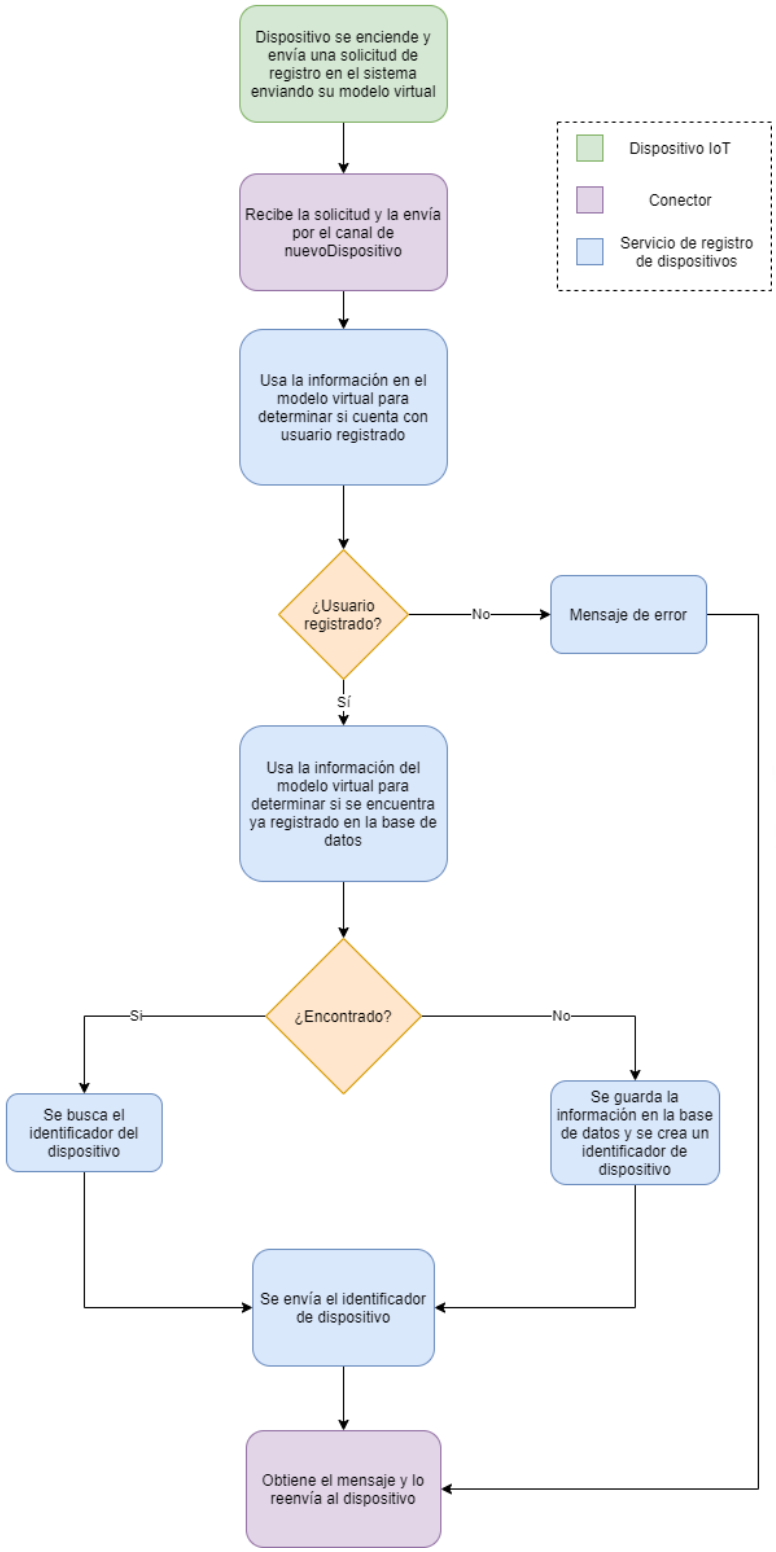


Figura 3-4 Diagrama de flujo de registro de dispositivo

Fuente: Elaboración propia

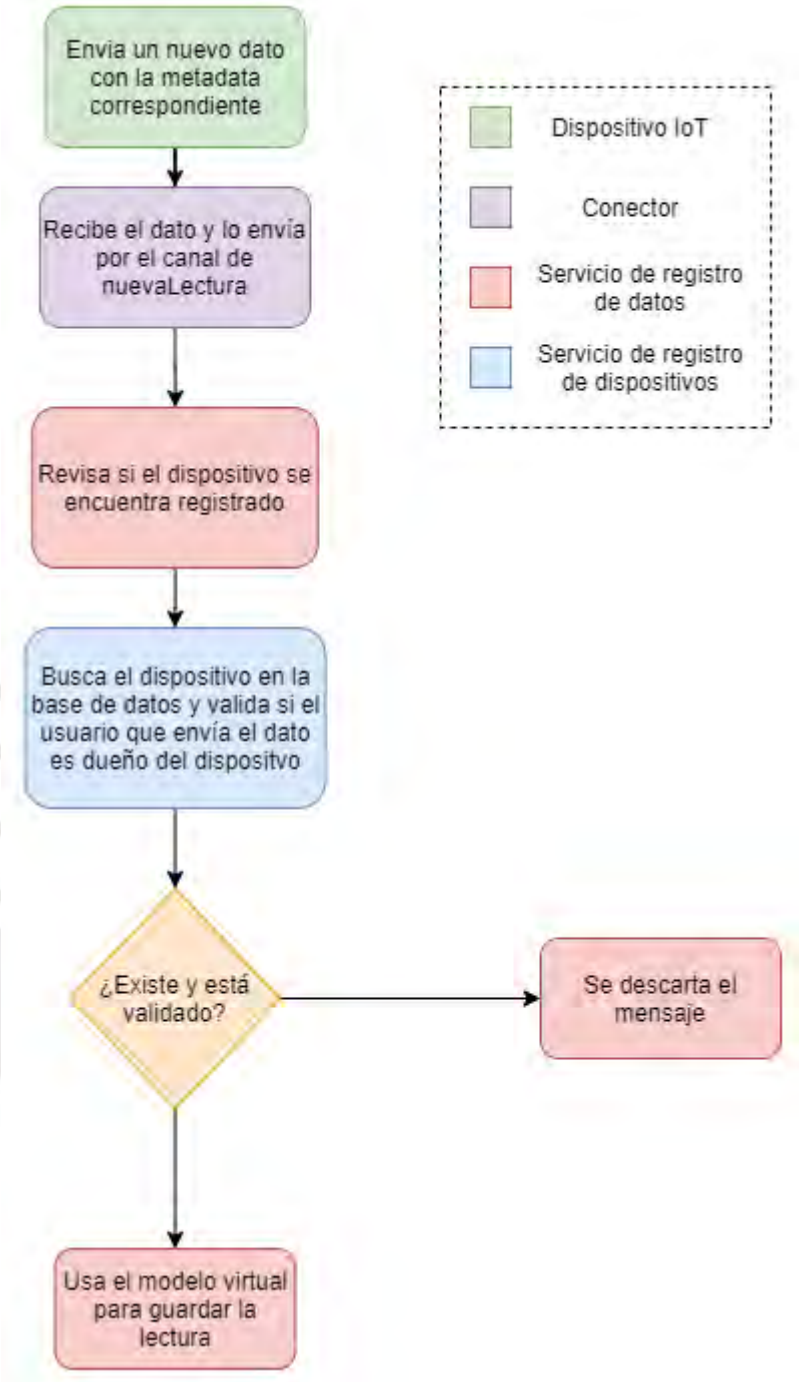


Figura 3-5 Diagrama de flujo de envío de datos

Fuente: Elaboración propia

Se procederá a explicar cada uno de los componentes y servicios que conforman la aplicación.

3.2. Diseño e implementación de todos los servicios y componentes

3.2.1. RabbitMQ

3.2.1.1. Conceptos básicos

Este broker es una parte integral de la aplicación por que cumple dos funciones importantes, es el sistema de comunicación entre los servicios basados en eventos y también es el broker para los dispositivos AMQP. Aunque en el diagrama lógico el conector AMQP sea una entidad diferente, esa función también será parte de este componente, ya que se permitirá que los dispositivos IoT se comuniquen directamente con él y realicen las acciones de registro y envío de datos. Aparte, como también cuenta con un plugin que permite manejar el protocolo MQTT [57], podría usarse como el broker y conector para ese protocolo.

RabbitMQ, y AMQP en general, funcionan por medio de un sistema de *exchanges, queues* y *bindings*, para recibir y enviar mensajes. Para poder entender mejor el diseño realizado se explicará cada uno de estos componentes:

Un e*xchange* es una entidad a la que los productores envían los mensajes y, de acuerdo al tipo de e*xchange* y metadata que se envía en el mismo mensaje (*routing key*), es que se encaminan a su destino. Para ello, los consumidores declaran *queues* que son entidades que reciben y guardan el mensaje para que puedan ser enviados a la aplicación deseada. Por otro lado, los *bindings* son entidades que se encargan de relacionar los *exchanges* con las *queues* mediante una *binding key* para enviar los mensajes de acuerdo al tipo de *exchange* con el que se cuente. Estos tipos son: *direct, fanout, topic y header* donde los usados para el middleware son los dos primeros. La diferencia entre ellos es que en *direct* el mensaje se envía a las colas donde el *routing* key y el *binding key* sean exactamente iguales, mientras que en *fanout*, el mensaje se envía a todas las colas con las que se encuentra enlazado [58], [59].

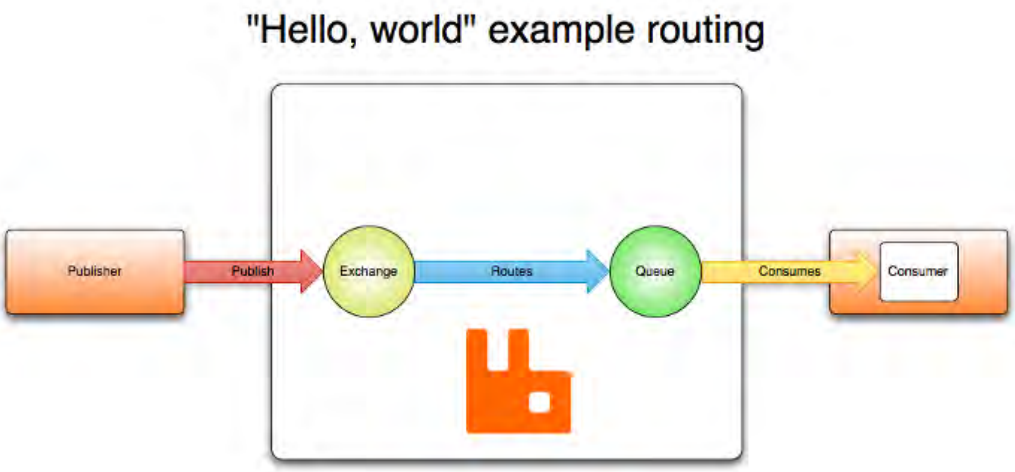


Figura 3-6 Ejemplo simple de los componentes de un sistema de comunicación AMQP

Fuente: [58]

3.2.1.2. Uso y despliegue en la aplicación

Como se observa en la Figura 3-1 y en los diagramas de flujo, se cuenta con dos canales de comunicación basados en eventos llamados “nuevoDispositivo” y “nuevaLectura”, los cuales emiten mensajes que van a llegar a servicios diferentes y funcionan de manera diferente.

El primero enlaza a los conectores con el servicio de registro de dispositivo, enviándose el mensaje de registro del dispositivo y devolviéndose un el ID de objeto que se genera cuando se guarda en la base de datos. Como se requiere de una comunicación bidireccional, se usa un modelo de *RPC (Remote Call Procedure)* donde el conector envía el mensaje a un *exchange* directo con información adicional, el nombre de una *queue* de respuesta y un ID de correlación que permita identificar el mensaje que inició la llamada. Cuando el servicio termina de procesar el registro, usa esta información para poder enviar el mensaje de respuesta y el conector usa el ID de correlación para enlazar el mensaje enviado con el dispositivo que realizó la petición. Una figura de ejemplo de un sistema RPC se muestra a continuación.

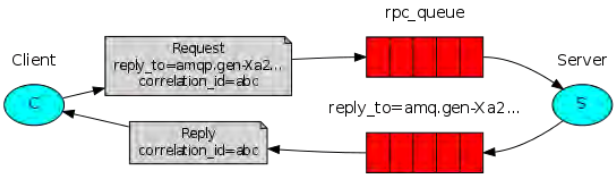


Figura 3-7 Ejemplo de RPC en AMQP

Fuente: [60]

Por otro lado, el canal de comunicación de "nuevaLectura" usa un sistema mucho más simple ya que no requiere de comunicación bidireccional. Cuando un mensaje llega al conector, este lo envía por un *exchange fanout* y será procesado por el servicio correspondiente. Si existe algún error al momento de procesarlo, el mensaje se descarta y se guarda el registro del error. Una ventaja de usar un *exchange* del tipo *fanout* es que este sistema puede extenderse con servicios adicionales, como uno de streaming de datos que permita a los usuarios obtener la información que envían sus dispositivos en tiempo real u otro de notificaciones cuando lleguen valores fuera de límites definidos.

En cuanto a la configuración, se usará la configuración por defecto ofrecida por la imagen Docker desarrollada por la organización que desarrolla el broker en su versión *3.8.5-management-alpine* con usuarios simples. Esto es posible porque la configuración de las entidades dentro del broker (*queues*, *bindings* y *exchanges*) se realiza de forma programática por cada uno de los servicios.

3.2.1.3. Alternativas de despliegue

Existen en el mercado diferentes despliegues e implementaciones que brindan este broker como un servicio, como es el caso de CloudAMQP, donde se puede alquilar cierta cantidad de nodos de diferentes características (en cuanto a número de conexiones, número de mensajes por segundos, número de colas, etc.) brindándose credenciales de acceso y una interfaz de administración [61]. Aunque en el middleware que se va a desarrollar en esta tesis se contará con una instancia del broker desplegada y administrada como parte de la aplicación, este servicio en la nube es una alternativa completamente viable para despliegues de producción que pueda realizarse de la aplicación, ya que contar con un clúster RabbitMQ correctamente configurado y

desplegado a una escala grande demandaría un equipo dedicado solo a su mantenimiento, mientras que CloudAMQP brinda soporte y se encarga completamente del mantenimiento del clúster cuando se compran sus servicios.

3.2.2. Conector HTTP

Como este servicio requiere conectarse con una gran cantidad de dispositivos y sea rápido en procesar las solicitudes se decidió implementarlo en el lenguaje Go, uno que tiene una alta velocidad de procesamiento y genera ejecutables con poco peso, útil si se quiere desplegar varias instancias de este servicio de forma rápida y con poco consumo de memoria.

Como se van a procesar paquetes HTTP y mensajes AMQP, se eligieron dos librerías para esa función: Gin-Gonic como framework web HTTP y TurboCookedRabbit para la parte de conexión AMQP. Junto con esas librerías, especialmente Gin, se descargan otras que permiten la validación de datos JSON como si fuera JSON Schema y otras funcionalidades más que permiten el desarrollo completo del servicio. Otra característica importante de Gin, y que también está presente en muchos otros frameworks de Go, es el uso de las gorutinas para el manejo de las solicitudes, eliminando en gran medida las limitaciones de usar un protocolo de comunicación síncrono como HTTP, manifestándose mejor mientras más solicitudes por segundo existan [62], [63].

En cuanto al funcionamiento del servicio, este expone dos rutas en su interfaz API Rest, una para el registro de nuevos dispositivos una en la ubicación "/*register_new_device*" y otra para nuevos datos en "/*new_reading*". Como se mencionó anteriormente, la primera ruta valida que los mensajes JSON enviados cumplan el esquema de mensajes de registro, ocurriendo un error si no es así, y de igual forma con la ruta de nuevas lecturas.

En cuanto a la conexión con RabbitMQ, TurboCookedRabbit permite contar con una conexión robusta con el broker y la creación de *exchanges, queues* y *bindings* de forma sencilla, y agregarles las funciones necesarias para el envío, recepción y procesamiento

de cada uno de los mensajes, como es el de nuevo registro y la obtención del *objID*, así como el envío del mensaje de nueva lectura.

Finalmente, se construyó la imagen Docker con el ejecutable compilado y se subió al repositorio público de Docker llamado DockerHub. De igual forma, el código de este servicio se encuentra en GitHub y su diseño de API con OpenAPI en SwaggerHub.

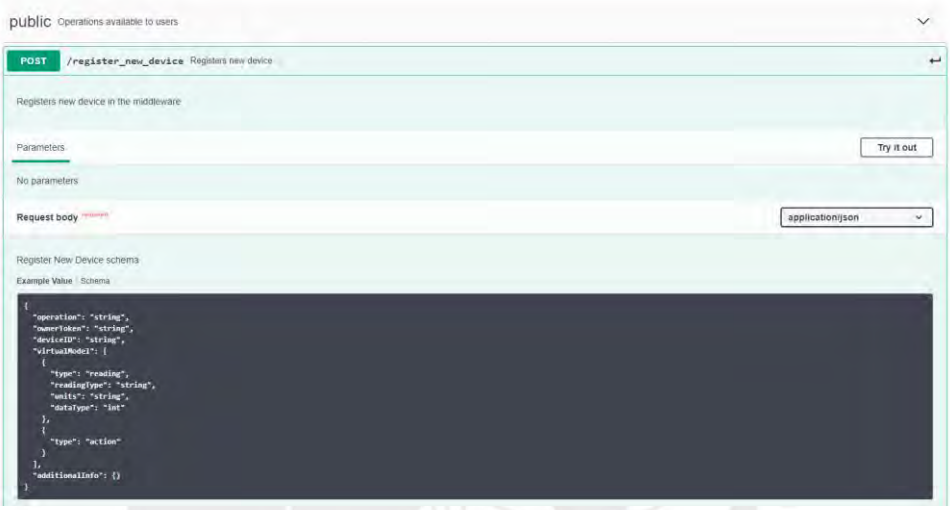


Figura 3-8 Documentación del ConectorHTTP en OpenAPI

Fuente: Elaboración propia

3.2.3. MongoDB

Base de datos NoSQL que permite guardar entradas con características completamente homogéneas, a diferencia de una base de datos SQL que requiere de una estructura fija para la información que se desee insertar.

En MongoDB, cada una de las entradas tiene el nombre de documento y cuenta la estructura de datos BSON, similar a JSON, pero de forma binaria. Otra característica usada de esta base de datos, es que permite contar con un validador de los documentos insertados para asegurar que los datos ingresados sean correctos, funcionando de forma similar a JSON Schema y usando una sintaxis similar. No es completamente estricto como la estructura de una tabla en una base de datos SQL, pero permite a los desarrolladores la garantía de que los datos ingresados cuentan, como mínimo, con campos obligatorios [64].

Para el despliegue se usará una imagen Docker proporcionada por el equipo de desarrolladores en su versión *4.2.8*, con la creación de usuarios por defecto y la inserción de los esquemas de validación correspondientes.

3.2.4. Servicio de registro de dispositivos

Este servicio está programado en Python por ser un servicio más complejo que el anterior y requiere de más experiencia en el lenguaje de programación por la lógica necesaria para su correcto funcionamiento. Como se describió anteriormente, se usará el framework FastAPI para la creación del servidor web, permitiendo que se escriba código de forma completamente asíncrona usando las características de Python descritas en secciones anteriores. Además, brinda muchas facilidades al momento de crear el servicio, como documentación autogenerada OpenAPI del API Rest de acuerdo al código, y cuenta con librerías como dependencias que permiten la fácil validación de los mensajes JSON y trabajo con las peticiones HTTP. Por otro lado, también cuenta con la capacidad de usar tokens OAuth2 para la autenticación de los usuarios al momento de realizar las consultas de recursos personales usando el servicio de usuarios y autenticación que se describirá en secciones siguientes [65]. Como se usa una framework asíncrono, las librerías que se usan para la conexión con la base de datos MongoDB y el broker RabbitMQ tienen que ser asíncronos para obtener el máximo rendimiento. Por lo tanto, se usarán las versiones asíncronas de las librerías presentadas anteriormente, identificables porque generalmente cuentan con el prefijo “aio” de *Asyncronous IO* como lo es aio-pika para AMQP y motor para MongoDB [66], [67].

En cuanto al funcionamiento del servicio, este es el consumidor de los eventos del canal “nuevoDispositivo”, por lo que, en la llegada de un mensaje, se valida el contenido del mensaje JSON enviado por el dispositivo, se valida que el “*ownerToken*” exista en el servicio de usuarios, se convierte al modelo virtual final quitándole el campo “*operation*” y se guarda en MongoDB, donde se genera el “*objID*” que será enviado al Conector por el canal de comunicación asignado. Si ocurre algún error en todo el proceso, será capturado y enviado también al conector para que se envíe el mensaje de error correspondiente.

En la parte de su interfaz API Rest, este cuenta con tres rutas para acceder a la información de los dispositivos, dos de las cuales están protegidas mediante un Token OAuth2, una que permite obtener la información de un dispositivo en particular y otra que permite obtener todos los dispositivos registrados a nombre del usuario. Para ambas operaciones, se obtiene el nombre de usuario del token descrito anteriormente, con lo cual se consulta los "*ownerTokens*" disponibles y se valida si el usuario puede acceder al dispositivo o no. Finalmente, existe una tercera ruta no pública que verifica si el "*objID*" de un dispositivo es válido y se obtiene su modelo virtual, el cual es usado por el servicio de registro de datos para guardar los datos que llegan.

Finalmente, se construyó la imagen Docker del servicio con las librerías necesarias y el código fuente y se publicó en DockerHub. El código fuente de la aplicación se encuentra en GitHub y su diseño de API con OpenAPI en SwaggerHub.

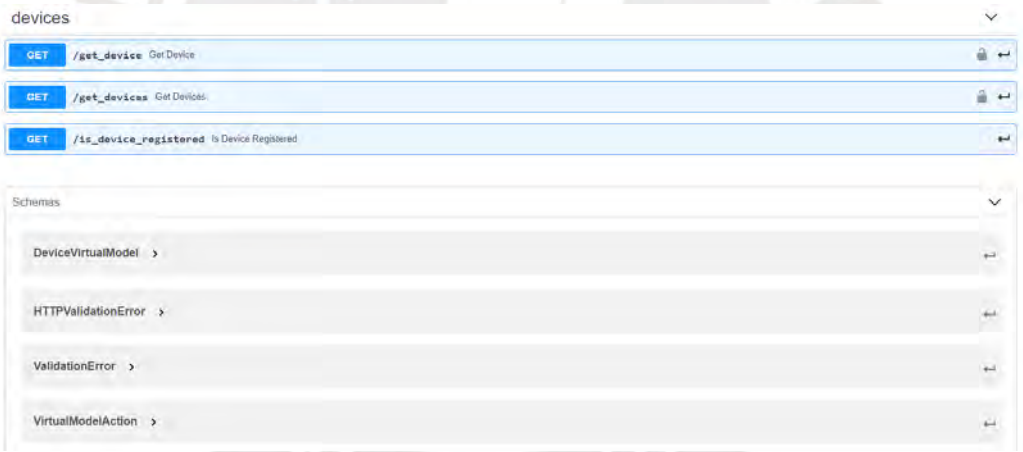


Figura 3-9 Documentación del servicio de Registro de Dispositivos en OpenAPI

Fuente: Elaboración propia

3.2.5. InfluxDB

Como se explicó anteriormente, InfluxDB es una *Time Series Database*, donde se guardan información orientada al tiempo, elegida porque se requiere guardar cada una de las lecturas realizadas por los sensores con una marca de tiempo o *timestamp* que identifique el momento donde se realizó la lectura. Esta base de datos cuenta con 3 conceptos importantes que deben ser explicados: *field, tag* y *measurement.*

- Un valor de campo o *field* es aquel que guarda un dato completamente cambiante y que no será usado por la base de datos para la indexación porque sería totalmente ineficiente. Por ejemplo, las medidas de un sensor de temperatura imaginario pueden variar entre -10 °C a 30 °C con varios decimales, por lo que sería ineficiente buscar todas las entradas alguna medición que sea exactamente 22.5 °C.

- Un valor de etiqueta o *tag* es aquel que guarda un dato poco variante y altamente repetible que puede usarse para la indexación y búsqueda porque suelen representar conjuntos de entradas. Por ejemplo, se pueden tener sensores desplegados en el distrito de San Miguel, por lo que todos esos sensores cuentan con la etiqueta "san-miguel" y se pueden obtener todas las entradas que cuenten con esa etiqueta de forma eficiente.

- Una medida o *measurement* es una forma de separar un grupo de entradas dentro de la base de datos, funcionando de la misma forma que una tabla en una base de datos SQL.

Para el despliegue de esta base de datos se usará la imagen Docker desarrollada por la organización que implementa InfluxDB en su versión *1.8-alpine*, sin configuración adicional más allá de generar un usuario y contraseña al momento de crear el contenedor.

3.2.6. Servicio de registro de datos

Al igual que el servicio anterior, este se encuentra desarrollado en Python usando el framework FastAPI, y también usa la librería *aio-pika* para la conexión con el broker RabbitMQ. Sin embargo, como este servicio requiere conectarse con una base de datos InfluxDB de forma asíncrona, se usa la librería *aioinflux* para esta función [68].

En cuanto al funcionamiento del servicio, este recibe los nuevos datos en el canal "nuevaLectura", donde con el "*objID*" y "*ownerToken*" puede consultar el modelo virtual al servicio de registro de dispositivos y validar si se cuentan con los permisos para guardar esta medición. Luego, con el modelo virtual y el valor de "*index*" de cada

medición enviada, se obtiene el nombre de la medición llamada "*readingType*", el cual será usado como valor de campo o *field* en InfluxDB. Como campos de etiqueta o *tag* se usarán el "*deviceID*" y "*objID*", para finalmente usar el "*ownerToken*" como nombre de tabla o *measurement*.

En la parte del API Rest se cuenta con una sola ruta protegida que permite obtener los datos guardados de cierto dispositivo en un rango de tiempo determinado. Para ello se envía el "*deviceID*" del dispositivo buscado, el rango de fechas y el "*readingType*" que se desea consultar. El servicio se encargará de verificar si el dispositivo pertenece al usuario de acuerdo al nombre de usuario presente en el Token OAuth2 y luego realizará una búsqueda en InfluxDB devolviendo los resultados obtenidos.

Finalmente, se construyó la imagen Docker y se publicó en DockerHub. El código fuente se encuentra disponible en GitHub y su diseño de API con OpenAPI en SwaggerHub.

Figura 3-10 Documentación del servicio de Registro de Datos en OpenAPI

Fuente: Elaboración propia

```
name: asdfoiuzxodiuqwerjokpqdf
time                device_id            humidity obj_id                   temperature
----                ---------            -------- ------                   -----------
1595318029281232000 network/group/device          5f128c0891b0e6f43129491c 20
1595318075735566000 network/group/device          5f128c0891b0e6f43129491c 20
1595318518874871000 network/group/device1 67      5f16a0deb0a357a3e4602ce7
1595318720616360000 network/group/device1 67      5f16a0deb0a357a3e4602ce7
1595318722574033000 network/group/device          5f128c0891b0e6f43129491c 20
>
```

Figura 3-11 Ejemplo de almacenamiento en InfluxDB por el servicio

Fuente: Elaboración propia

3.2.7. Servicio de usuarios y autenticación

Para el servicio de usuarios y autenticación se cuenta con dos sub aplicaciones que cumplen el rol de gestionar y autenticar a los usuarios. Uno es el servicio de gestión de gestión de usuarios en sí, mientras que otro es Hydra. Se procederá a explicar ambos componentes.

3.2.7.1. Hydra

Esta aplicación, escrita en Go, es una implementación del protocolo OAuth2 que permite desplegar un servidor de autorización para la autenticación mediante este protocolo. Con el servidor ya ejecutándose, se crean las credenciales de cada una de las aplicaciones y ya es posible realizar la autenticación de los usuarios y clientes usando cualquiera de los diferentes *authentication flows* definidos en el estándar. En el caso del middleware, solo se usará el *authorization flow* para obtener un JWT que contenga el nombre de usuario autenticado. Esta aplicación requiere como dependencia una base de datos SQL para el guardado de las sesiones de los tokens, pero también puede usarse en el modo de memoria para que guarde la información hasta el siguiente reinicio [69].

Una particularidad de esta aplicación, es que solo se encarga de la emisión de los tokens de acceso dejando la implementación del proveedor de identidades y el sistema de inicio de sesión y autorización de permisos o *scopes* al equipo de desarrollo que lo despliegue. Por lo tanto, el servicio de usuarios será encargado de esa función y será descrito a continuación.

Para el despliegue se usará la imagen Docker proporcionada por los desarrolladores en su versión *v1.6* y la configuración necesaria para que use el servicio de usuarios como su proveedor de identidades e inicio de sesión.

3.2.7.2. Servicio de usuarios

Para este servicio se decidió usar Java como lenguaje de programación, usando el framework Spring Boot con las librerías *Webflux* como servidor web, y *Mongo Reactive* como conector con la base de datos MongoDB. Estas dos librerías, y el paradigma de programación que se siguió, sigue el paradigma de programación reactiva que se explicó

anteriormente, proporcionando mejores similares a la programación asíncrona de Python.

Este servicio expone una gran cantidad de rutas en su API Rest, siendo tres las usadas por Hydra para realizar la autenticación y entrega de tokens, y otras cinco para la administración de usuarios dentro del sistema. Entre esas rutas de administración se cuenta con rutas para el registro de nuevos usuarios, consulta de la información de un usuario, generación de "*ownerToken*", etc. En general, las rutas no cuentan con esquemas complejos y serán detallados en el diseño del API en OpenAPI.

Finalmente, se construyó la imagen Docker y se publicó en DockerHub. El código fuente se encuentra en GitHub, así como el diseño en OpenAPI en SwaggerHub.

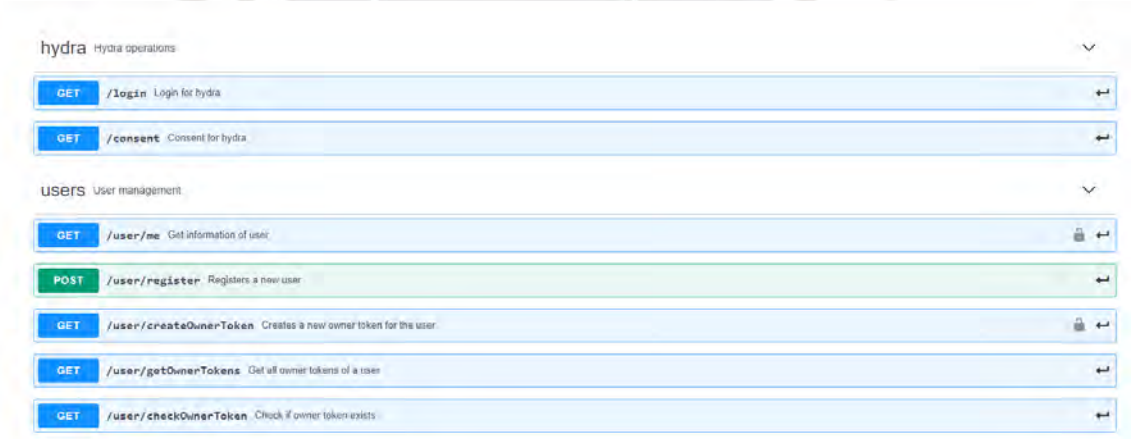


Figura 3-12 Documentación del servicio de Usuarios en OpenAPI

Fuente: Elaboración propia

3.2.8. API Gateway

Para el API Gateway se optó por usar la librería Spring Cloud Gateway, que permite crear una aplicación de este tipo de forma completamente sencilla, solamente otorgando un archivo con las rutas a las cuales se tiene que re direccionar. Esto es posible ya que no se requiere de lógica compleja para la conexión de los usuarios a los servicios, solo se necesita que se reenvíen las solicitudes.

La configuración aplicada es la de contar con sub rutas que apunten a cada uno de los servicios que presentan una interfaz de usuario y se reenvía la solicitud que llega al

Gateway. Por ejemplo, si se quiere registrar un usuario en el servicio de usuarios, a la solicitud original de "*/user/register*" se le agrega la sub ruta "/users" por lo que termina como "*/users/user/register*" y esa solicitud se realiza al *API Gateway* en vez de al servicio en sí. Sin embargo, todo eso es configurable y puede cambiarse al gusto del usuario.

Como los servicios anteriores, se construyó la imagen Docker y se publicó en DockerHub. De igual forma el código fuente se encuentra en GitHub.

3.3. Despliegue en Docker Compose

Para el despliegue de la aplicación se usará Docker Compose, la herramienta que permite especificar el despliegue de varios servicios en un archivo llamado *docker-compose.yml* y permite la configuración de cada uno de los servicios como si se trataran de contenedores normales. Este archivo y las configuraciones de cada uno de los servicios del middleware se encuentran en GitHub.

A continuación, se presentan tablas y diagramas de la disposición final de los servicios, los archivos de configuración con los que cuentan, y los puertos configurados y expuestos por defecto.

Tabla 3-1 Archivos de configuración de los servicios

Servicio	Archivo	Uso
Conector-HTTP	configuration.env	Variables de configuración del servicio
Gateway	application.yml	Archivo de configuración de Spring
Usuarios	application.properties	Archivo de configuración de Spring
Registro-datos	configuration.env	Variables de configuración del servicio
	prestart.sh	Script que espera a que InfluxDB y RabbitMQ se encuentren disponibles para iniciar el servicio

Tabla 3-1 Archivos de configuración de los servicios

Registro-dispositivos	configuration.env	Variables de configuración del servicio
	prestart.sh	Script que espera a que MongoDB y RabbitMQ se encuentren disponibles para iniciar el servicio
Rabbit	configuration.env	Variables de configuración del broker
Mongo	configuration.env	Variables de configuración del servicio
	restoreDatabase.sh	Script que inicializa la base de datos con los esquemas de validación y de índices necesarios para el correcto funcionamiento del middleware
	dump/	Carpeta con los archivos que usa el script anterior para inicializar la base de datos
Influx	configuration.env	Variables de configuración del servicio
Hydra	configuration.env	Variables de configuración del servicio
	.hydra.yml	Archivo de configuración de Hydra

Fuente: Elaboración propia

La configuración que se encuentra por defecto sirve para un despliegue local porque la configuración de Hydra valida que los tokens de acceso generados funcionen solo para *localhost*. Si se quiere desplegar la aplicación en la nube, se requiere cambiar el archivo de configuración de Hydra y cambiar la dirección de emisión de tokens por la de la máquina donde se encuentre, así como las direcciones del inicio de sesión y consentimiento. También existe un archivo llamado *create_clients.sh* que crea los clientes autorizados a generar tokens, donde también tiene que cambiarse las direcciones a donde se envían los tokens luego de un inicio de sesión exitoso.

Otra configuración que sería sensible cambiar es el de las credenciales de acceso a los diferentes servicios, sobre todo si el despliegue se va a realizar en una nube pública con acceso a Internet. Esto se debe a que las credenciales configuradas para las bases de datos y RabbitMQ son simples y descifrables, permitiendo que una persona no autorizada acceda a los recursos del middleware.

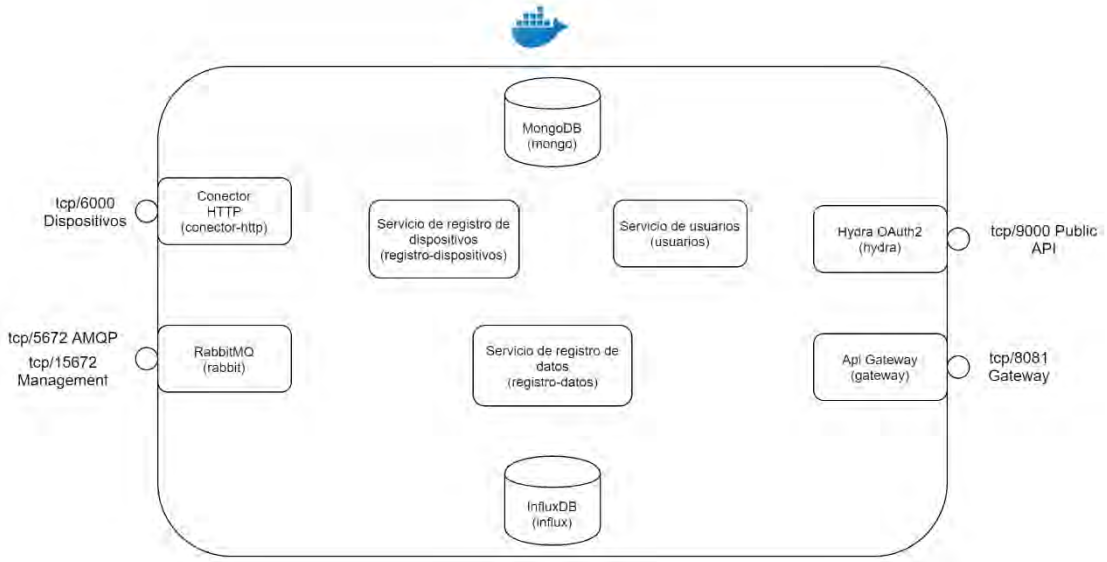


Figura 3-13 Diagrama de contenedores desplegados con Docker Compose

Fuente: Elaboración propia

Como se observa en la figura, solo cuatro servicios exponen sus puertos fuera de la red virtual creada para el despliegue. Los dos de la derecha son usados por los usuarios para acceder a sus recursos creados en el middleware, mientras que los dos de la derecha son usados por los dispositivos para el registro y envío de datos. Aun así, es posible cambiar esta configuración en el archivo *docker-compose.yml* y especificar que otros servicios exponen sus puertos. Por ejemplo, podría exponerse el puerto de MongoDB para administrar los dispositivos y usuarios del sistema usando un GUI hecho para esa base de datos.

Capítulo 4. Análisis del plan de pruebas

En este capítulo se presentará el análisis de las pruebas a realizar, su justificación con los objetivos planteados en el Capítulo 1 y las herramientas necesarias para realizarlas.

4.1. Pruebas del sistema

Luego de realizado el diseño y la implementación del sistema, se tienen consideradas las siguientes pruebas para verificar que se hayan cumplido los objetivos planteados en el Capítulo 1:

- Pruebas al flujo total del sistema, desde la creación de un usuario, el registro de un dispositivo y sus datos, y finalmente la consulta de todos sus recursos guardados en el *middleware*. Se comprueba el correcto funcionamiento de todos los sistemas y la consistencia de la información guardada. Esta prueba cumple tres objetivos específicos, el primero porque demuestra que un dispositivo IoT puede enviar información de forma estandarizada independiente de sus características o protocolos gracias al uso de modelos virtuales y conectores; cumple el segundo protocolo específico, porque demuestra que la información

enviada por los dispositivos se almacena en una base de datos para su posterior consulta; finalmente, cumple parcialmente el tercer objetivo específico porque demuestra la integración de todos los servicios bajo una arquitectura de microservicios.

- Pruebas de capacidad de procesamiento de registro de datos de los dispositivos donde se medirá el tiempo de respuesta del sistema, el número de peticiones por segundo que aguanta antes de presentar fallos y la integridad de la información luego de terminada la prueba. De la misma forma, se realizarán pruebas escalando los servicios para determinar el porcentaje de mejora se genera por el escalamiento de los servicios, cumpliendo la parte faltante del tercer objetivo específico. Como se requiere enviar un número alto de peticiones al sistema, se usará una herramienta de prueba de carga que permita simular peticiones de registro de datos al sistema, el cuál será presentado más adelante.

- Prueba de despliegue y uso del middleware oM2M comparando el flujo de funcionamiento y, de ser posible, el rendimiento con el middleware implementado en esta tesis, cumpliendo el cuarto objetivo específico.

4.2. Herramientas a usar

4.2.1. Postman

Una aplicación que permite enviar peticiones Rest u otros protocolos como SOAP o GraphQL permitiendo definir todos los parámetros necesarios de la consulta, como es el cuerpo de la petición, las cabeceras y más. También soporta la autenticación de diversas formas, siendo una de ellas OAuth2, necesaria para probar el acceso a recursos protegidos del middleware. Finalmente, permite visualizar las respuestas y toda la metadata correspondiente de forma sencilla [70].

Esta herramienta será usada para todas las pruebas que requieren interactuar con una interfaz API Rest, siendo la primera prueba el caso con mayor grado de utilización porque todas las peticiones realizadas serán mediante esta interfaz que presentan el *API Gateway* y el conector HTTP.

La versión usada será el cliente de Windows en su versión v7.29.1 ejecutándose en la computadora de desarrollo y conectada a Internet.

4.2.2. Locust.io

Herramienta que permite hacer pruebas de carga en un API escribiendo peticiones en Python y ejecutándolas de forma concurrente midiendo el tiempo de respuesta, el número de peticiones exitosas y fallidas, mostrando gráficos de la evolución de la prueba. Todas las peticiones se escriben en Python, permitiendo una gran flexibilidad por el gran número de librerías disponibles para realizar consultas sumamente complejas y con diversos protocolos [71].

Como las pruebas se escriben en código, se encontrarán disponibles en GitHub para la revisión, así como los resultados obtenidos de cada una de ellas.

Para el despliegue de las pruebas se usará la imagen Docker proporcionada por los desarrolladores de la herramienta en su versión 1.1 y usando Docker Compose para levantar varios contenedores, aprovechando que cuenta con la opción de funcionar de forma distribuida.

4.3. Entorno de pruebas

Como el despliegue del *middleware* implementado en esta tesis solamente requiere de una máquina que tenga instalado Docker y Docker Compose, se optó por realizar las pruebas en la nube de AWS, específicamente en una instancia EC2 de las siguientes características:

Tabla 4-1 Características de la instancia EC2 de prueba

Tipo de instancia EC2	t2.large
vCPU	2
Memoria RAM	8 GB
Almacenamiento	30 GB
Dirección IP pública	34.234.229.191
Nombre de host	ec2-34-234-229-191.compute-1.amazonaws.com

Sistema Operativo	Ubuntu Server LTS 18.04 (x64)
Versión de Docker	19.03.12 (build 48a66213fe)
Versión de Docker Compose	1.24.0 (build 0aa59064)

Fuente: Elaboración propia

En esta instancia se desplegarán el middleware completo, los contenedores de Locust.io para las pruebas y una instancia del middleware oM2M para realizar las comparaciones respectivas.

4.4. Impacto del sistema

En esta sección se comentará brevemente el impacto que causa este sistema en diferentes ámbitos.

4.4.1. Impacto ambiental

Al tratarse de un sistema digital y en fase de prueba de concepto, no puede medirse el impacto ambiental directamente. Sin embargo, podrá medirse cuando se implementen proyectos que usen el *middleware* como en ciudades inteligentes o en zonas agrarias, donde el uso de IoT ayude a la medición y consumo responsable de los recursos naturales.

4.4.2. Impacto económico

Igualmente, que en el impacto ambiental, no es posible medir directamente el impacto económico a menos que sea usado en un proyecto. Aun así, se predice que la facilidad de despliegue de este sistema en hardware poco potente permitirá que los proyectos existentes cuenten con mayor flexibilidad en su diseño y presupuesto.

4.4.3. Impacto social

En cuanto al impacto social, gracias a este sistema se acercarán tecnologías IoT al público general como lo son las casas y ciudades inteligentes. Proyectos como sistemas de regado inteligentes, medición de calidad de aire, automatización, entre otros lograrán que se mejore la calidad de vida de las personas.

4.4.4. Impacto cultural

Finalmente, el impacto cultural tampoco es posible de medir directamente en el estado actual del *middleware*. Sin embargo, un mayor uso de IoT generará que sea más común el uso de tecnología y medios digitales en más ámbitos de la vida cotidiana.

Capítulo 5. Evaluación del desempeño y comparación con middlewares alternativos

Este capítulo presenta los resultados de las pruebas realizadas, describiendo y analizando cada uno de los casos. Finalmente, se hará una comparación con el middleware oM2M en cuanto a funcionamiento y despliegue.

5.1. Pruebas realizadas

5.1.1. Prueba de flujo de funcionamiento

Esta primera prueba muestra el flujo que seguirá un usuario del *middleware*, mostrando los pasos necesarios lograr registrar un dispositivo y el envío de los datos. Las fotos de los resultados se encuentran en el siguiente repositorio GitHub.

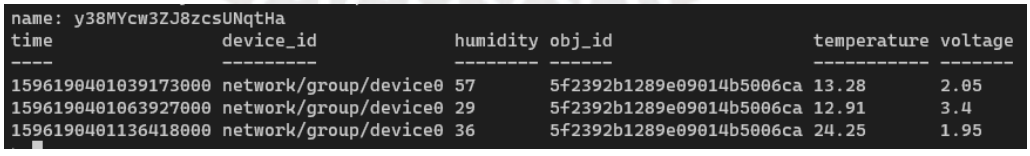
a. En primer lugar, un usuario se registra en la aplicación realizando una petición POST a la ruta "*/user/register*" del servicio de usuarios con el cuerpo descrito en su documentación de OpenAPI. El sistema enviará una respuesta con el usuario guardado en la base de datos o con un mensaje de error.

b. Contando con credenciales en el sistema, el usuario obtiene un token de acceso OAuth2 para realizar las peticiones siguientes. En el caso de la prueba se realiza mediante Postman y se obtiene el JWT conteniendo, entre otros datos, el nombre de usuario para identificar a la entidad que realiza la consulta.

c. Para registrar un dispositivo se requiere de un "*ownerToken*" el cual se obtiene realizando una petición GET a la ruta "*/user/createOwnerToken*" del servicio de usuarios y se obtiene como respuesta el nuevo token o un mensaje de error.

d. Se forma el modelo virtual del dispositivo, tomando en cuenta las características que tiene el dispositivo, y se agrega la información necesaria, usando el token obtenido en el paso anterior para relacionar al dispositivo con el usuario al que le pertenece. Esta información se envía al Conector correspondiente y se recibe como respuesta el "*objID*" del dispositivo o un mensaje de error.

e. Se envía el mensaje de guardado de datos, usando el ID obtenido en el paso anterior, donde se envían las mediciones tomadas y los datos adicionales extra. En este caso siempre se obtiene un mensaje de envío correcto, a menos que exista un problema dentro de la aplicación en sí. No existe confirmación hacia el dispositivo que los datos enviados se guardaron correctamente. Sin embargo, si no existe error con los datos enviados, estos se guardarán correctamente en la base de datos.



```
name: y38MYcw3ZJ8zcsUNqtHa
time                device_id             humidity obj_id                   temperature voltage
----                ---------             -------- ------                   ----------- -------
1596190401039173000 network/group/device0 57       5f2392b1289e09014b5006ca 13.28       2.05
1596190401063927000 network/group/device0 29       5f2392b1289e09014b5006ca 12.91       3.4
1596190401136418000 network/group/device0 36       5f2392b1289e09014b5006ca 24.25       1.95
```

Figura 5-1 Muestra del guardado correcto de los datos en InfluxDB

Fuente: Elaboración propia

f. Con los pasos anteriores, el usuario realizar consultas sobre sus recursos, como la información de todos sus dispositivos realizando una petición GET a la ruta "*/getDevice*" del servicio de registro de dispositivos, u obtener la información de todos sus dispositivos realizando una petición GET a la ruta "*/get_devices*" del

mismo servicio. También se puede realizar consultas de los datos enviados por los dispositivos, enviando una petición POST a la ruta "*/get_device_data*" del servicio de registro de datos con el cuerpo especificado en la documentación OpenAPI.

Como se observa, el flujo de funcionamiento es sencillo y consta de seis pasos para registrar y enviar los datos recolectados por un dispositivo, de los cuales los primeros 2 requieren intervención de un usuario y los demás pueden automatizarse e implementarse en todos los dispositivos IoT deseados. De la misma forma, la consulta de la información disponible puede realizarse y usarse por aplicaciones desarrolladas por el usuario porque se realizan mediante Rest, la cual cuenta con implementaciones y librerías en los lenguajes más populares de programación.

Por temas de la pandemia no se pudo realizar esta prueba con dispositivos IoT reales, por lo que el armado de los modelos virtuales y la información enviada tuvo que inventado para su realización. Sin embargo, el funcionamiento no varía ya que se tendría que realizar los mismos pasos con los dispositivos de generar en código los modelos virtuales y de envío de información. Lo que cambia es el flujo de como programa el sistema de recolección de datos y las llamadas al *middleware*, pero eso queda fuera del desarrollo del sistema descrito en esta tesis.

Con esta prueba se comprueba el funcionamiento completo de la aplicación y se verifica que cumple con los objetivos planteados de poder registrar dispositivos de distintas características y el almacenamiento de sus datos enviados.

5.1.2. Pruebas de capacidad de procesamiento

Esta segunda prueba mide la capacidad que tiene el sistema de procesar peticiones de guardado de datos simultáneas, midiendo el tiempo de respuesta, número de peticiones fallidas y la consistencia de la información en la base de datos luego de la petición.

Se realizaron cuatro iteraciones de esta prueba, desde el mejor caso donde el número de peticiones por segundo es bajo hasta una prueba con un alto número de peticiones y se observó degradamiento del tiempo de respuesta y uso de recursos de hardware.

Esta prueba se realizó con la herramienta Locust.io, donde los archivos para cada una de las pruebas se encuentran en GitHub, junto con los resultados obtenidos para cada uno de ellos. Las imágenes de los resultados de cada una de las pruebas también se encuentran disponibles en el Anexo 1.

Como línea base, se toma una medición del uso de recursos de todos los servicios en reposo:

Tabla 5-1 Uso de recursos de hardware en reposo

Servicio	Uso de CPU	Uso de memoria
Conector HTTP	0%	11.23 MiB
Registro de datos	0.21%	98.17 MiB
Registro de dispositivos	0.22%	119 MiB
Usuarios	0.08%	379.7 MiB
Gateway	0.10%	330.4 MiB
MongoDB	0.23%	72.78 MiB
InfluxDB	0.03%	57.29 MiB
RabbitMQ	1.21%	120.1 MiB
Hydra	0.00%	13.86 MiB

Fuente: Elaboración propia

Como se observa, el uso de recursos es mínimo especialmente de los servicios escritos en el lenguaje Go con un uso de memoria menos a 20MiB. Esto muestra la gran eficiencia de recursos con las que cuenta este lenguaje.

5.1.2.1. Primera prueba de carga

En la primera prueba de este tipo solo se simularon 40 usuarios simultáneos donde se guardó la información de un solo dispositivo. El número de dispositivos guardados es importante ya que el servicio de registro de datos consulta con el servicio de registro de dispositivos el modelo virtual del dispositivo el cual va a almacenar sus datos. Si siempre es el mismo dispositivo, la respuesta que envíe puede ser usada del caché de la petición anterior reduciendo el tiempo de procesamiento.

Tabla 5-2 Resultados de la primera prueba de carga

Número de peticiones	3403
Número de peticiones por segundo	12.46
Tiempo de respuesta promedio	2.65 ms
Tiempo de respuesta mínimo	1.76 ms
Tiempo de respuesta máximo	38.32 ms
Número de peticiones con error	0
Datos almacenados en InfluxDB	3403

Fuente: Elaboración propia

Como se observa, el *middleware* superó esta prueba sin ningún problema porque no existió pérdida de datos y el uso de hardware durante la prueba aumentó menos de 1% por cada servicio. Otro dato interesante que se observa es el tiempo de respuesta menor a los 10ms, el cual se debe a que el Conector solo tiene que enviar el evento al canal de "nuevaLectura" y recibir la confirmación del broker para responder al dispositivo.

5.1.2.2. Segunda prueba de carga

Para la segunda prueba se aumentó el número de usuarios simultáneos a 200 con el registro de datos de solo 1 dispositivo. Presentándose los siguientes resultados:

Tabla 5-3 Resultados de la segunda prueba de carga

Número de peticiones	12418
Número de peticiones por segundo	303.67
Tiempo de respuesta promedio	1.37 ms
Tiempo de respuesta mínimo	0 ms
Tiempo de respuesta máximo	21 ms
Número de peticiones con error	0

Datos almacenados en InfluxDB	12419

Fuente: Elaboración propia

Este resultado subió el número de peticiones de 12 a más de 300 por segundo, lo que causó que el servicio de registro de datos se vea superado en capacidad de procesamiento y dejo gran parte de los mensajes en el broker. Sin embargo, esto no fue problema ya que Rabbit está diseñado para funcionar de esa manera, enviando los datos restantes a la velocidad que podía consumirlo el servicio, que era aproximadamente 100 mensajes por segundo.

Otro dato interesante que se muestra es que en la base de datos apareció un resultado más que el número de peticiones enviadas. Esto puede deberse a algún bug que existe en la implementación del servicio de guardado o de comunicación con el broker pues también se observará en las siguientes pruebas, pero no ha sido identificado completamente.

Aun así, el tiempo de la aplicación permaneció bajo e incluso presentó respuestas menos a 1 ms, lo que demuestra la velocidad que tiene Go para el procesamiento de datos. Cosa que se demuestra con el consumo de memoria y CPU, donde el primero subió a 20 MiB y el segundo a 5%.

5.1.2.3. Tercera prueba de carga

La diferencia de esta prueba con la anterior fue que se enviaron datos de tres dispositivos, lo que dificultará el guardado de la información ya que las respuestas de caché serán menos frecuentes, disminuyendo la capacidad de procesamiento del servicio de registro de datos. Se presentaron los siguientes resultados:

Tabla 5-4 Resultados de la tercera prueba de carga

Número de peticiones	14943
Número de peticiones por segundo	316.22
Tiempo de respuesta promedio	4.68 ms

Tiempo de respuesta mínimo	1.60 ms
Tiempo de respuesta máximo	57.26 ms
Número de peticiones con error	0
Datos almacenados en InfluxDB	14943

Fuente: Elaboración propia

A pesar de contar con tres dispositivos, el resultado de ambos servicios fue similar al caso anterior, donde la respuesta del conector fue rápida y la capacidad de procesamiento del servicio de registro de datos se mantuvo en alrededor de 100 mensajes por segundo, como se observa en la siguiente figura:

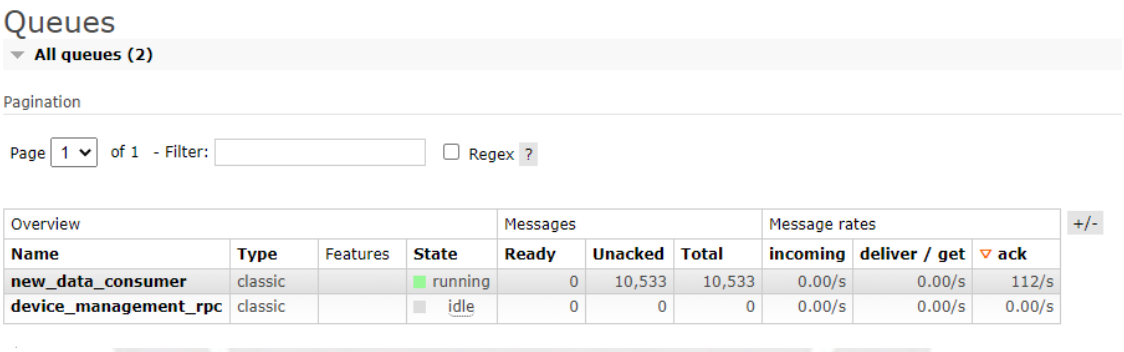


Figura 5-2 Procesamiento de los mensajes en la prueba de carga

Fuente: Elaboración propia

Al igual que la segunda prueba, esto significa que el guardado de los mensajes toma más tiempo antes de procesarse completamente, pero esto no es un problema ya que solo se requiere el guardado de la información y no acceder a ella en tiempo real.

5.1.2.4. Cuarta prueba de carga

Para la última prueba se aumentó considerablemente el número de usuarios que realizan peticiones, aumentando también el número de peticiones por segundo y ya se mostró degradamiento notable de la experiencia de uso del *middleware*. Para esta prueba también se escaló el número de instancias del servicio de registro de datos, en preparación de la subida de carga que iba a existir. Se presentan los resultados:

Tabla 5-5 Resultados de la cuarta prueba de carga

Número de peticiones	55879
Número de peticiones por segundo	496.05
Tiempo de respuesta promedio	1720.42 ms
Tiempo de respuesta mínimo	1.66 ms
Tiempo de respuesta máximo	13123.94 ms
Número de peticiones con error	0
Datos almacenados en InfluxDB	57167

Fuente: Elaboración propia

Como se observa, el tiempo promedio de respuesta subió hasta casi 2 segundos, teniendo una solicitud que demoró 13 segundos en procesarse, pero no se produjeron fallos en el envío del mensaje al broker. En cuanto a la capacidad de procesamiento del servicio de registro de datos, esta no mejoro en lo absoluto a pesar de haber escalado el número de instancias a 2. Esto puede deberse a que los recursos de hardware ya que se observó una utilización del 100% del procesador entre todos los servicios y lo que explicaría el bajo rendimiento del conector.

Como curiosidad, es nota que el número de datos guardados en InfluxDB resulta mucho mayor al número de peticiones enviadas, lo cual es un problema que se debe encontrar la causa y solucionar para un mejor funcionamiento ante una carga alta.

En general, se puede concluir que con el hardware probado de 2 procesadores y 8 GB de memoria se puede obtener un rendimiento de cerca de 300 peticiones por segundo donde RabbitMQ funciona como buffer para el guardado de los mensajes y el servicio de registro de datos logra consumirlo a una tasa de 100 mensajes por segundo. Se elige esta cantidad de peticiones por los resultados de la tercera y cuarta prueba, donde se observa que, a partir de las 400 peticiones por segundo, el sistema empieza a degradarse de forma exponencial. Por lo tanto, se dimensiona el sistema dejando un porcentaje de los recursos para otros programas y sistema operativo que puedan estar operando en la

máquina usada. Este número podría incrementarse en servidores con mayores capacidades de hardware, pero demuestra que, en un entorno de bajas capacidades, el sistema rinde eficientemente y sin errores al momento de procesar la llegada de nuevos datos.

5.1.3. Comparación de funcionamiento con el middleware oM2M

El middleware oM2M cuenta con una sección de descarga y compilación en su página web, donde se encuentran los pasos requeridos para lograr correr el sistema. Sin embargo, luego de esas instrucciones no existe la documentación necesaria para el uso completo de la aplicación y el registro de dispositivos o aplicaciones, aparte de pequeños ejemplos que sirven poco para explicar el sistema completo.

Por tanto, se decidió descargar y compilar el código fuente para determinar si el sistema era lo suficientemente intuitivo como para que un usuario lo maneje sin la documentación requerida. Luego de la compilación y ejecutar la aplicación uno se encuentra con la siguiente pantalla:

```
Aug 01, 2020 3:21:59 AM ch.ethz.inf.vs.californium.network.config.NetworkConfig createStandardWithFile
INFO: Create standard properties with file Californium.properties
Aug 01, 2020 3:21:59 AM ch.ethz.inf.vs.californium.server.Server start
INFO: Starting server
Aug 01, 2020 3:21:59 AM ch.ethz.inf.vs.californium.network.CoAPEndpoint start
INFO: Starting Endpoint bound to 0.0.0.0/0.0.0.0:5683
INFO [ONEM2M.SDT] ctor
INFO [ONEM2M.SDT] Activation
INFO [ONEM2M.SDT] ctor
INFO [ONEM2M.SDT] Activation
osgi> ss
"Framework is launched."


id      State       Bundle
0       ACTIVE      org.eclipse.osgi_3.10.2.v20150203-1939
1       RESOLVED    javax.servlet_3.1.0.v201410161800
2       RESOLVED    javax.xml_1.3.4.v201005080400
3       RESOLVED    org.apache.commons.codec_1.6.0.v201305230611
4       RESOLVED    org.apache.commons.logging_1.1.1.v201101211721
                    Fragments=29
5       ACTIVE      org.apache.felix.gogo.command_0.10.0.v201209301215
6       ACTIVE      org.apache.felix.gogo.runtime_0.10.0.v201209301036
7       ACTIVE      org.apache.felix.gogo.shell_0.10.0.v201212101605
8       RESOLVED    org.apache.httpcomponents.httpclient_4.3.6.v201511171540
9       RESOLVED    org.apache.httpcomponents.httpcore_4.3.3.v201411290715
10      ACTIVE      org.eclipse.equinox.cm_1.1.0.v20131021-1936
11      ACTIVE      org.eclipse.equinox.console_1.1.0.v20140131-1639
12      ACTIVE      org.eclipse.equinox.ds_1.4.200.v20131126-2331
13      ACTIVE      org.eclipse.equinox.event_1.3.100.v20140115-1647
14      ACTIVE      org.eclipse.equinox.http.jetty_3.0.200.v20131021-1843
```

Figura 5-3 Pantalla de inicio de oM2M

Fuente Elaboración propia

Esta pantalla indica los módulos disponibles y cargados en el sistema, donde se observa que cuenta con librerías para trabajar con MQTT, CoAP y HTTP, además de una capa de almacenamiento con MongoDB. Sin embargo, no se pudo iniciar de ninguna forma un sistema que se encargara de registrar dispositivos o de acceder a los recursos dentro del sistema. Parte de esto se debe a que la documentación dentro del mismo programa es extensa y poco navegable, donde más de la mitad de ayuda que ofrece es más del sistema de consola de Java que del *middleware* en sí.

Luego de investigar el sistema por unas horas y de buscar información en Internet, se concluyó que la diferencia en el uso de este *middleware* con el desarrollado en capítulos anteriores es completamente diferente. Aunque este cuente con muchas más funcionalidades y sea más completo, la falta de documentación hace que su uso sea más dificultoso y complejo de desplegar. A esto se suma que el desarrollo de este sistema se detuvo a finales del 2018, contándose con modificaciones pequeñas al código en las últimas semanas, pero para actualizar temas de licencias de uso.

Por esta misma razón, no se realizaron comparativas de rendimiento ya que no se logró desplegar una instancia funcional del programa. Sin embargo, en [7, p. 6] se cuentan con mediciones hechas por el asesor para comparar el rendimiento de su *middleware* con oM2M, las cuales serán presentadas a continuación:

Tabla 5-6 Resultados de prueba secuencial

	oM2M	**Middleware asesor**
Tiempo medio (ms)	484.82	470.61
Tiempo mínimo (ms)	460	451
Tiempo máximo (ms)	720	675

Fuente: [7, p. 6]

Tabla 5-7 Prueba paralela con 10 peticiones por segundo

	Peticiones satisfactorias	**Peticiones fallidas**	**Mediana de tiempo de respuesta (ms)**	**Tiempo medio de respuesta (ms)**	**Peticiones por segundo**
oM2M	746	1295	310	1166	8.07

Middleware asesor	1290	1234	470	732	13.97

Fuente: [7, p. 6]

Tabla 5-8 Prueba paralela con 100 peticiones por segundo

	Peticiones satisfactorias	**Peticiones fallidas**	**Mediana de tiempo de respuesta (ms)**	**Tiempo medio de respuesta (ms)**	**Peticiones por segundo**
oM2M	86	998	410	864	2.89
Middleware asesor	1314	22	470	479	156.86

Fuente: [7, p. 6]

Comparando estos resultados con los obtenidos en la sección 5.1.2, se observa que los tiempos de respuesta del sistema propuesto son mejores en, aproximadamente, 400 ms y a 200 peticiones por segundo mayores. Esto es posible gracias al uso de eventos y comunicación asíncrona como fue descrito en el diseño del sistema. Aun así, sería recomendable realizar un despliegue correcto de oM2M para realizar nuevas comparaciones.

Conclusiones

- Se logró implementar un *middleware* que permite la integración horizontal de dispositivos IoT de diferentes características gracias al uso de modelos virtuales para su caracterización. Además, se cuenta con un sistema que permite la consulta de la información ya guardada y existiendo una separación entre los recursos de diversos usuarios.

- Este sistema se implementó bajo una arquitectura de microservicios, permitiendo que la extensibilidad de funciones del sistema sea mucho más sencilla ya que no requiere conocer completamente el funcionamiento de todos los servicios, si no solo aquellos con los que se necesita interactuar y es posible desarrollarlo en el lenguaje de programación que se desee, como es el caso de los servicios ya implementados donde se usaron tres lenguajes: Go, Python y Java.

- Se realizaron pruebas de rendimiento y de capacidad de carga al sistema, donde se concluyó que para un hardware que cuente con 2 CPUs y 8GB de memoria RAM, se obtiene que el número de peticiones por segundo que es capaz de procesar es de 300 por segundo con un tiempo de respuesta promedio menores a 5 ms. Si se aumenta la tasa de peticiones, el tiempo de respuesta se degrada de forma casi exponencial.

- Aunque la prueba no demostró una mejora significativa del escalamiento de los servicios, se demostró que cada uno de ellos puede escalar de forma independiente de acuerdo a la capacidad de carga que se requiera en un instante determinado de tiempo.

- No se pudo realizar un despliegue del *middleware* oM2M para las comparaciones, pero se usó como referencia los resultados obtenidos por el asesor en su trabajo de tesis. Se observó que los tiempos de respuesta son mejores en más 400 ms y funciona correctamente con un mayor número de

peticiones por segundo. Por lo tanto, se concluye que el sistema propuesto es mejor en cuanto a capacidad de procesamiento de los datos.

Recomendaciones y trabajos futuros

Desarrollar un sistema bajo una arquitectura de microservicios es complicado por el número de distintos sistemas con los que se tiene que trabajar y más aún cuando el equipo de desarrollo es de una sola persona. Eso limitó completamente el alcance final que tuvo que tener el sistema, y por lo que se cuenta con un amplio número de puntos de mejora, entre los que se encuentran:

- Desarrollar un servicio de *streaming* de datos en tiempo real, lo que permita a las aplicaciones acceder de forma instantánea a las mediciones que realizan los dispositivos IoT.

- Uso de diferentes protocolos de comunicación como HTTP/2, HTTP/3, GraphQL, los cuales tienen diversas ventajas sobre HTTP/1 que es el usado en este trabajo.

- Desarrollar un servicio de notificaciones que alerte a los usuarios cuando las lecturas de un dispositivo no se guarden correctamente.

- Realizar pruebas de rendimiento frente a *middlewares* alternativos.

Bibliografía

[1] Oracle, “What Is the Internet of Things (IoT)?”, *Oracle*. https://www.oracle.com/internet-of-things/what-is-iot.html (consultado dic. 05, 2019).

[2] C. McClelland, “What is IoT? - A Simple Explanation of the Internet of Things”, *IoT for all*, ene. 09, 2020. https://www.iotforall.com/what-is-iot-simple-explanation/ (consultado ene. 15, 2020).

[3] Statista Research Department, “IoT: number of connected devices worldwide 2012-2025”, *Statista*, nov. 27, 2016. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ (consultado dic. 05, 2019).

[4] P. Smedley, “IPv4, IPv6, and the IoT”, *Connected World*, jul. 09, 2016. https://connectedworld.com/ipv4-ipv6-and-the-iot/ (consultado mar. 20, 2020).

[5] J. Benavides, “Middleware Design for Application Integration in IoT Networks”, University of Campinas, Campinas, 2020.

[6] T. Yokotani y Y. Sasaki, “Comparison with HTTP and MQTT on required network resources for IoT”, en *2016 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, sep. 2016, pp. 1–6, doi: 10.1109/ICCEREC.2016.7814989.

[7] J. Benavides y H. E. Hernandez-Figueroa, “Middleware Design for Application Integration in IoT Networks”, en *Proceedings - 2017 International Conference on Computational Science and Computational Intelligence, CSCI 2017*, dic. 2018, pp. 1326–1331, doi: 10.1109/CSCI.2017.231.

[8] Microsoft Team, “Non-relational data and NoSQL”, *Azure Architecture Center*, dic. 02, 2018. https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data (consultado may 13, 2019).

[9] InfluxDB Team, “Time Series Database (TSDB) Explained”, *InfluxDB*. https://www.influxdata.com/time-series-database/ (consultado mar. 09, 2020).

[10] C. Richardson, “Microservice Architecture pattern”, *Microservices.io*. https://microservices.io/patterns/microservices.html (consultado mar. 20, 2020).

[11] A. Gerber y J. Romeo, “Connecting all the things in the Internet of Things”, *IBM Developer*, mar. 27, 2017. https://developer.ibm.com/technologies/iot/articles/iot-lp101-connectivity-network-protocols/ (consultado may 02, 2020).

[12] A. Macedo, “Uso de una arquitectura basada en eventos como capa de comunicación para microservicios”, Pontificia Universidad Católica del Perú, 2020.

[13] Elastic Team, “What is Elasticsearch? | Elastic”, *Elastic*. https://www.elastic.co/what-is/elasticsearch (consultado may 20, 2020).

[14] S. Balakrishnan, “Microservices — Myths and Misunderstandings”, *DZone Integration*, mar. 14, 2016. https://dzone.com/articles/microservices-myths-and-misunderstanding (consultado jun. 20, 2020).

[15] Microsoft Team, “¿Qué es DevOps? Explicación de DevOps”, *Microsoft Azure Documentation*. https://azure.microsoft.com/es-mx/overview/what-is-devops/#overview (consultado jun. 20, 2020).

[16] RedHat Team, “¿Qué son la integración/distribución continuas (CI/CD)?”, *RedHat*. https://www.redhat.com/es/topics/devops/what-is-ci-cd (consultado jun. 26, 2020).

[17] AWS Team, “¿Qué es la entrega continua?”, *AWS*. https://aws.amazon.com/es/devops/continuous-delivery/ (consultado jun. 20, 2020).

[18] AWS Team, “¿Qué es la integración continua?”, *AWS*. https://aws.amazon.com/es/devops/continuous-integration/ (consultado jun. 20, 2020).

[19] Swagger Team, “OpenAPI Specification - Version 3.0.3 | Swagger”, *Swagger*. https://swagger.io/specification/ (consultado mar. 15, 2020).

[20] RabbitMQ Team, “Messaging that just works — RabbitMQ”, *RabitMQ*. https://www.rabbitmq.com/ (consultado may 26, 2020).

[21] Python Team, “Coroutines and Tasks”, *Python*. https://docs.python.org/3/library/asyncio-task.html (consultado abr. 20, 2020).

[22] Reactor Team, “Project Reactor”, *Project Reactor*. https://projectreactor.io/ (consultado abr. 20, 2020).

[23] MongoDB, “What Is MongoDB?”, *MongoDB*. https://www.mongodb.com/what-is-mongodb (consultado dic. 08, 2019).

[24] T. Sandholm, “InfluxDB: The Good, the Bad, and the Ugly”, *Medium*, sep. 06, 2017. https://medium.com/@thomas.sandholm/influxdb-the-good-the-bad-and-the-ugly-b409f8c370a3 (consultado jun. 20, 2020).

[25] J. K., “All About MongoDB NoSQL Database: Advantages and Disadvantages”, *Acodez*, jul. 30, 2019. https://acodez.in/mongodb-nosql-database/#Advantages_and_Disadvantages_of_MongoDB (consultado jun. 20, 2020).

[26] M. Anicas, “An Introduction to OAuth 2”, *DigitalOcean*. https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2 (consultado jul. 06, 2020).

[27] Docker, “What is a Container?”, *Docker*. https://www.docker.com/resources/what-container (consultado dic. 08, 2019).

[28] E. Boersma, “Docker Image vs Container: Everything You Need to Know”, *Stackify*, may 03, 2019. https://stackify.com/docker-image-vs-container-everything-you-need-to-know/ (consultado jun. 10, 2020).

[29] Kubernetes Team, “¿Qué es Kubernetes?”, *Kubernetes*. https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/ (consultado ene. 20, 2020).

[30] pav, “Dissecting Kubernetes example”, *Dots and Brackets Code Blog*, nov. 15, 2017. https://codeblog.dotsandbrackets.com/kubernetes-example/ (consultado jun. 15, 2020).

[31] NGINX Team, “Service Discovery in a Microservices Architecture”, *NGINX*. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ (consultado jun. 15, 2020).

[32] N. Bhide, “K8s Know-How: Service Discovery and Networking”, *DZone Microservices*, ene. 19, 2019. https://dzone.com/articles/k8s-knowhow-service-discovery-and-networking (consultado abr. 20, 2020).

[33] Hashicorp Team, “Consul by HashiCorp”, *Consul*. https://www.consul.io/ (consultado jun. 15, 2020).

[34] A. Z., “Overview of Top 5 Languages to Build Microservices”, *RubyGarage*, abr. 11, 2019. https://rubygarage.org/blog/top-languages-for-microservices (consultado jun. 20, 2020).

[35] M. Zaforas, “Cómo implementar microservicios con Python en minutos”, *Paradigma Digital*, 2017. https://www.paradigmadigital.com/dev/implementar-microservicios-python-minutos/ (consultado jun. 22, 2020).

[36] Spring Team, “Microservices”, *Spring*. https://spring.io/microservices (consultado jun. 22, 2020).

[37] A. Aslam, “Micro — a microservices toolkit - Micro”, *Medium*, jul. 11, 2016. https://medium.com/microhq/micro-a-microservices-toolkit-c403145b65c1 (consultado jun. 22, 2020).

[38] A. Pourafshar, “Publish-Subscribe Security in IoT Environments”, *SecurityCompass*. https://resources.securitycompass.com/blog/publish-subscribe-security-in-iot-environments-2# (consultado jun. 20, 2020).

[39] Eclipse Foundation, “Eclipse Mosquitto”. http://mosquitto.org/ (consultado dic. 08, 2019).

[40] RabbitMQ Team, “Which protocols does RabbitMQ support?”, *RabbitMQ*. https://www.rabbitmq.com/protocols.html (consultado jun. 20, 2020).

[41] JSON, “Introducción a JSON”. https://www.json.org/json-es.html (consultado dic. 08, 2019).

[42] ETSI, “Our one Machine-to-Machine Partnership Project (oneM2M)”, *ETSI*. https://www.etsi.org/committee/1419-onem2m (consultado dic. 08, 2019).

[43] oneM2M, “oneM2M Service Layer”, *oneM2M*. https://www.onem2m.org/getting-started/onem2m-overview (consultado dic. 08, 2019).

[44] ETSI, “ETSI M2M solution introduction”, may 2014. Consultado: jun. 20, 2020. [En línea]. Disponible en: https://www.etsi.org/images/files/events/2014/201405_dgconnect_smartm2m appliances/etsi_m2m_introduction_main.pdf.

[45] A. Deol, K. Figueredo, S.-W. Lin, B. Murphy, D. Seed, y J. Yin, “Advancing the Industrial Internet of Things”, Peter Klement (XMPro, dic. 2019. Consultado: jun. 27, 2020. [En línea]. Disponible en: https://www.onem2m.org/images/files/IIC_oneM2M_Whitepaper_final_2019_12_12.pdf.

[46] oneM2M Team, “oneM2M: Solving the IoT Platform Challenge”, nov. 2015. Consultado: jun. 20, 2020. [En línea]. Disponible en: https://www.onem2m.org/images/files/onem2m-executive-briefing_A4.pdf.

[47] M. Mccool, “OCF Overview Including Summary of New Features in OCF 1.0 Candidate Draft”, 2017. Consultado: jun. 20, 2020. [En línea]. Disponible en: https://www.w3.org/2017/05/wot-f2f/slides/OCF_Overview_for_W3C_F2F_McCool.pdf.

[48] BusinessWire Team, “El Open Interconnect Consortium ayuda a los desarrolladores a abordar el Internet de las cosas con un nuevo kit de herramientas para desarrolladores”, *BusineesWire*, feb. 11, 2016. https://www.businesswire.com/news/home/20160211005677/es/ (consultado jun. 20, 2020).

[49] OCF Team, “3D Printer Resource Model Example”, *oneIoTa*, ene. 12, 2020. https://www.oneiota.org/revisions/6134 (consultado jun. 20, 2020).

[50] Y. Wang, L. Wei, Q. Jin, y J. Ma, “Alljoyn Based Direct Proximity Service Development: Overview and Prototype”, en *2014 IEEE 17th International Conference on Computational Science and Engineering*, dic. 2014, pp. 634–641, doi: 10.1109/CSE.2014.138.

[51] F. Liu, “AllJoyn Framework An Internet of Things Software Protocol”.

[52] Eclipse, “What is Eclipse OM2M?”, *Eclipse OM2M*. https://www.eclipse.org/om2m/ (consultado dic. 08, 2019).

[53] M. Ben Alaya, Y. Banouar, T. Monteil, C. Chassot, y K. Drira, “OM2M: Extensible ETSI-compliant M2M service platform with self-configuration capability”, en *Procedia Computer Science*, ene. 2014, vol. 32, pp. 1079–1086, doi: 10.1016/j.procs.2014.05.536.

[54] IoTivity Project, “IoTivity / iotivity-lite”, *Gitlab*.

https://gitlab.iotivity.org/iotivity/iotivity-lite#building-sample-applications-on-linux (consultado jun. 20, 2020).

[55] JSON Schema Team, “JSON Schema | The home of JSON Schema”, *JSON Schema*. https://json-schema.org/ (consultado jul. 20, 2020).

[56] JSON Schema Team, “Implementations”, *JSON Schema*. https://json-schema.org/implementations.html (consultado jul. 20, 2020).

[57] RabbitMQ Team, “MQTT Plugin — RabbitMQ”, *RabbitMQ*. https://www.rabbitmq.com/mqtt.html (consultado jul. 20, 2020).

[58] RabbitMQ Team, “AMQP 0-9-1 Model Explained — RabbitMQ”, *RabbitMQ*. https://www.rabbitmq.com/tutorials/amqp-concepts.html (consultado jul. 20, 2020).

[59] CloudAMQP Team, “Part 4: RabbitMQ Exchanges, routing keys and bindings - CloudAMQP”, *CloudAMQP*. https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html (consultado jul. 20, 2020).

[60] RabbitMQ Team, “RabbitMQ tutorial - Remote procedure call (RPC)”, *RabbitMQ*. https://www.rabbitmq.com/tutorials/tutorial-six-python.html (consultado jul. 20, 2020).

[61] CloudAMQP Team, “CloudAMQP - RabbitMQ as a Service”, *CloudAMQP*. https://www.cloudamqp.com/ (consultado jul. 20, 2020).

[62] Gin-gonic Team, “gin-gonic/gin”, *Github*. https://github.com/gin-gonic/gin#benchmarks (consultado jul. 14, 2020).

[63] Houseofcat, “TurboCookedRabbit”, *Github*. https://github.com/houseofcat/turbocookedrabbit (consultado jul. 14, 2020).

[64] MongoDB Team, “Schema Validation — MongoDB Manual”, *MongoDB*.

https://docs.mongodb.com/manual/core/schema-validation/ (consultado jul. 20, 2020).

[65] S. Ramirez, “FastAPI”, *FastAPI*. https://fastapi.tiangolo.com/ (consultado jul. 28, 2020).

[66] aio-pika Team, “aio-pika Documentation”, *ReadtheDocs.io*. https://aio-pika.readthedocs.io/en/latest/ (consultado jul. 21, 2020).

[67] Motor Team, “Motor: Asynchronous Python driver for MongoDB”, *ReadTheDocs.io*. https://motor.readthedocs.io/en/stable/ (consultado jul. 21, 2020).

[68] aioinflux Team, “aioinflux’s Documentation”, *ReadTheDocs.io*. https://aioinflux.readthedocs.io/en/stable/ (consultado jul. 21, 2020).

[69] ORY Team, “ORY Hydra | ORY Hydra”, *Ory.sh*. https://www.ory.sh/hydra/docs/ (consultado jul. 21, 2020).

[70] Postman Team, “Postman API Client”, *Postman*. https://www.postman.com/product/api-client/ (consultado jul. 15, 2020).

[71] J. Heyman, C. Byström, J. Hamrén, y H. Heyman, “What is Locust?”, *Locust*. https://docs.locust.io/en/stable/what-is-locust.html (consultado dic. 08, 2019).