Máster en Ingeniería y Ciencia de Datos

Diseño e implementación de una plataforma IoT para monitorización de flotas de dispositivos heterogéneos

Autor:
Juan Díaz Suárez

Tutor:
Agustín Carlos Caminero Herraez

Curso 2020/2021

Convocatoria de julio

Resumen

En este Trabajo Fin de Máster se propone una arquitectura para una plataforma IoT capaz de escalar y gestionar multitud de dispositivos, a la par de permitir la conexión con herramientas Big Data como Apache Spark o Hadoop. Se muestra además una implementación simplificada preparada para ser desplegada en un clúster de Kubernetes y que cuenta además con un *dashboard* para su monitorización.

Esta memoria comienza con una introducción a los dispositivos IoT y con un repaso de las plataformas IoT disponibles actualmente. A continuación se define el concepto de Big Data y las formas más comunes de procesarlo: arquitecturas Lambda y Kappa.

Se sigue con una presentación de arquitectura genérica para una plataforma IoT, diferenciando entre componentes esenciales y complementarios. Tras ello se define la arquitectura que se implementa en este trabajo. A continuación se presentan unos casos de uso que buscan remarcar la utilidad de la plataforma y lo sencillo que sería usarla como núcleo de un producto mayor.

Se dedica después una sección a entender cómo se ha creado la plataforma, incluyendo una introducción a Kubernetes y al resto de herramientas que se han utilizado, además de presentar el código desarrollado.

Finalmente se realiza una evaluación de la plataforma que busca comprobar su correcto funcionamiento, el de las herramientas de monitorización y del escalado del clúster.

Palabras clave

Plataformas IoT, Big Data, Kubernetes, Kafka, Go.

Índice general

Índice de figuras

Índice de cuadros

Listings

Capítulo 1

Introducción, motivación y objetivos

1.1. Introducción

El Internet de las Cosas (Al-Fuqaha et al., 2015), también llamado IoT por sus siglas en inglés, es una tecnología de reciente aparición y rápidamente cambiante (Jerome Henry, 2017). Aunque no recibe una definición clara, principalmente se entiende como una red de dispositivos interconectados tanto de forma inalámbrica como por cable. Estas 'cosas' (Kortuem et al., 2010), se conectan a su vez a Internet para realizar dos funciones principales: almacenar los datos que recogen en un almacenamiento externo y ser operados a distancia. A partir de esta visión, los dispositivos se pueden categorizar como sensores y actuadores respectivamente.

Las redes IoT están formadas por sensores de telemetría, dispositivos de *tracking* de pedidos o dispositivos más generales como aparatos *e-health*, objetos personales como relojes y pulseras inteligentes, teléfonos móviles o instrumentos de domótica(Patel et al., 2016).

Para poder gestionar estas redes se han diseñado las plataformas IoT, encargadas de la ingesta de la información, la teleoperación de dispositivos y almacenamiento de los datos, entre otras funcionalidades. Estas plataformas suelen ofrecer a su vez integraciones con aplicaciones de análisis de datos, visualización del estado de la red o creación de alertas para configurar envíos por email si las medidas de ciertos sensores superan un umbral.

La mayoría de proveedores cloud (Google, Azure o AWS, por ejemplo) ofrecen solu-

ciones más o menos completas para gestión de dispositivos IoT. Además, disponen de recursos más específicos que permitirían crear una plataforma IoT basada solamente en componentes cloud, pero exigiría una dedicación mayor para el diseño y el despliegue.

1.2. Motivación y objetivos

El objetivo principal de este trabajo es diseñar una arquitectura para una plataforma IoT. Se busca además que permita conectar herramientas de procesamiento Big Data como pudieran ser las herramientas de Apache[1]. Por otro lado, se ha desarrollado para que sea *cloud agnostic*, es decir que se pueda desplegar tanto en servidores físicos como en proveedores cloud y que además escale con facilidad.

La necesidad surge de que la gran variedad de plataformas IoT que existen actualmente (Guth et al., 2018) están orientadas a ser un producto final y modificarlo para satisfacer las necesidades del proyecto que se quiera crear implicaría hacer modificaciones sobre su código. Se quiere proponer una plataforma que sirva como núcleo de proyectos mayores, que escale y que se pueda desplegar tanto en nube privada como pública.

1.3. Estructura de la memoria

Para alcanzar ese objetivo se parte de un estudio previo (sección 2) en el que se define el concepto de IoT y su historia. Se definen los requisitos que debe cumplir una plataforma IoT para considerarse como tal. Se exploran las soluciones comerciales actuales y cuáles son los componentes de los principales proveedores cloud, GCP (*Google Cloud Platform*), AWS (*Amazon Web Services*) y Azure, que se podrían utilizar para construir una. Como la que se propone debe ser *cloud agnostic*, no se van a utilizar estos componentes pero es importante conocerlos y saber qué tecnologías utilizan, ya que por lo general suelen modificar herramientas de código libre como ocurre con Google PubSub, basado en Apache Kafka. Se reflexiona también sobre el sentido que tiene crear una plataforma en lugar de contratar una o modificar las opciones de código libre.

En este estudio previo se define también el Big Data y cómo se procesa. A continuación

[1] https://projects.apache.org/projects.html?category#big-data

se introducen dos de las arquitecturas más comunes: la arquitectura Lambda, que realiza un procesamiento en *batch* y otro en *streaming*, y la arquitectura Kappa, que es la que se va a utilizar en la implementación y que solamente utiliza la capa de *streaming*. Finalmente se detalla en profundidad la diferencia entre *streaming* y *batch* para poder comprender las dificultades de su procesamiento y la importancia de cada uno.

Una vez se han establecido las bases teóricas del trabajo se procede a definir una arquitectura para una plataforma IoT genérica, en la sección 3. En este punto habrá que distinguir entre los componentes esenciales, que se encuentran en todas las plataformas, y los opcionales. Se aprovecha también para definir las funciones de cada componente y mencionar algunas herramientas que podrían ser útiles para desarrollarlos.

A continuación se propone una arquitectura para la prueba de concepto que se desarrolla en este trabajo. Como cualquier otra plataforma IoT, utiliza los componentes fundamentales y se añade además uno para monitorizar y visualizar el estado de la plataforma. Se podrían incluir muchas más herramientas, pero se perdería el enfoque del trabajo que es comprender las funciones y los componentes de una plataforma IoT. En este mismo apartado se mencionan algunos casos de uso que se podrían crear a partir de este desarrollo, comprendiendo las áreas del marketing, la ciberseguridad y la industria, además de una comparación con las plataformas disponibles en el mercado.

Terminado el diseño, en la sección 4 se detalla la implementación. Se va a utilizar Kubernetes para orquestar los contenedores donde se despliega cada componente. Es una tecnología relativamente nueva, compleja y muy importante en este trabajo, así que se le dedica una detallada introducción. El resto de herramientas se explican también pero sin tanto nivel de detalle. Se ha creado un repositorio en Gitlab[2] con el código necesario para replicar la plataforma. Al ser una prueba de concepto, no se necesitan apenas recursos informáticos para el despliegue, pero a su vez Kubernetes permite escalarla tanto como haga falta para soportar grandes cargas de trabajo, siempre y cuando se dispongan de suficientes nodos.

En la implementación se desarrolla una API REST para la recogida de datos y se envían a un clúster de Kafka, que se utiliza también para el almacenamiento. Se crea además un módulo de reglas para gestionar qué dispositivos pueden publicar datos. Por

[2] https://gitlab.com/iok8s

otro lado, Prometheus es el encargado de recoger las métricas de la API, del clúster de Kafka y del clúster de Kubernetes y representarlas en un *dashboard* de Grafana. Se ha escrito también un simulador de dispositivos en el lenguaje Go para poder importarlo desde cualquier desarrollo e ingestar dato en la plataforma, aunque con cualquier librería que genere peticiones HTTP serviría.

Finalmente, se realiza una serie de pruebas de carga para comprobar el correcto funcionamiento del clúster y de las herramientas de visualización.

1.4. Resumen del capítulo

En este capítulo se ha hecho una breve descripción de conceptos relacionados con los dispositivos IoT. Se explica también la motivación que hay detrás del trabajo y el objetivo que se busca. Se finaliza con un resumen de las distintas secciones que forman la memoria.

Capítulo 2

Estado del arte

2.1. Introducción

En esta sección se define el IoT y los componentes que lo forman. A continuación se detalla el concepto de 'plataformas IoT', en lo que se basa el grueso del trabajo, y los requisitos que deben cumplir. Esta explicación da pie a un breve resumen del estado actual de estas plataformas, incluyendo proyectos de código libre y otros empresariales. A lo largo de la sección se mencionan servicios y proveedores *cloud* que aparecerán a su vez entre las soluciones comerciales. Se cierra esta primera parte con una reflexión sobre las ventajas de diseñar frente a contratar una plataforma IoT.

En un segundo apartado, muy relacionado con el campo del IoT, se define el Big Data, cómo se procesa y cuáles son las arquitecturas más utilizadas para ello. Finalmente se entra en detalle sobre las dos formas de procesar datos: *streaming* y *batch*, y qué técnicas se usan en cada una de ellas.

2.2. Dispositivos y plataformas IoT

2.2.1. Definición

Se entiende por dispositivo IoT cualquier elemento que pueda enviar o recibir información. Ésta se puede enviar mediante Internet directamente o utilizar otro tipo de

comunicación, como podrían ser las ondas de radio, y que un dispositivo receptor (una pasarela) se encargue de transmitir la información a internet.

Estos dispositivos deben gestionarse desde sistemas externos ya que suelen formar redes de cientos o miles de sensores y actuadores. Para que un sistema se pueda considerar una plataforma IoT, debe ser capaz no solo de interconectar dispositivos heterogéneos, sino de gestionarlos, provisionar nuevos elementos, operar aquellos que lo permitan y almacenar la información recibida(Rayes and Salam, 2017). Sistemas más complejos deberán asegurar también que la transmisión de datos será segura y automatizar acciones en función de los datos que lleguen.

2.2.2. Historia

Los primeros dispositivos conectados, aunque no a internet, fueron sensores de telemetría. El primero se instaló en 1874 en el Mont Blanc, conectado mediante onda corta. Conceptos similares a las redes IoT los plantean científicos como Tesla, quien presentó la idea de un gran cerebro de objetos interconectados ya en 1926:

> *When wireless is perfectly applied, the whole Earth will be converted into a huge brain, which in fact it is, all things being particles of a real and rhythmic whole.*

El nombre de IoT fue propuesto por Kevin Ashton en 1999(Ashton, 2009) para referirse al proyecto en el que estaba inmerso en ese momento: una red de dispositivos de seguimiento en una cadena de suministro que utilizaban identificación mediante radiofrecuencia (RFID). Sin embargo, los primeros dispositivos conectados a Internet aparecieron a finales de los años 80, siendo el primero una tostadora ideada por John Romkey y Simon Hackett(Romkey, 2017).

Realmente el IoT aparece poco más tarde que Internet, como una consecuencia directa o un nuevo caso de uso. Gracias a la descentralización que permiten las redes WAN y MAN, surge la necesidad de controlar y gestionar multitud de dispositivos a distancia. No solamente se necesitaba teleoperarlos, sino que principalmente se buscaba recoger y analizar toda la información generada por éstos. Sin IoT, los dispositivos RFID debían

ser controlados desde distancias de pocos kilómetros. Una vez conectados a Internet, se puede recoger y tratar la información desde cualquier parte del mundo.

Los casos de uso del IoT crecen día a día, pero destacan la monitorización en tiempo real, mantenimiento predictivo, reducción de costes energéticos en edificios inteligentes e industria 4.0, logística y seguimiento de envíos, agricultura inteligente, *smart cities*...

2.2.3. Plataformas IoT

Ahora bien, el IoT gana mayor utilidad cuantos más dispositivos estén conectados a esa red y cuanto más sencillo sea de integrar con las tecnologías más utilizadas, para poder aplicar análisis a los datos recogidos o ejecutar tareas en función de estos datos. La solución que permite recoger, gestionar y analizar todos los datos recogidos por estos instrumentos son las plataformas IoT.

Éstas, al igual que el IoT, tampoco tienen una definición cerrada, pero su objetivo es permitir que aplicaciones y sistemas de alto nivel interactúen con los protocolos y métodos de comunicación de bajo nivel que controlan los dispositivos conectados a la red(Tamboli, 2019). Los requisitos que deben cumplir son los siguientes:

Escalabilidad La capacidad de cómputo y almacenamiento de la plataforma debe ser fácilmente ampliable para no limitar el número de dispositivos conectados de forma simultánea.

Fiabilidad Se debe asegurar que no habrá pérdida de información ni interrupción del servicio en caso de fallo en alguna de las máquinas. Habrá casos de uso, como vigilancia o medicina, donde la fiabilidad sea lo más importante y otros como la agricultura donde pueda quedar en un segundo plano.

Personalización Para poder soportar la velocidad de aparición de nuevas tecnologías, protocolos de comunicación o dispositivos, la plataforma debe poder editarse y ampliarse fácilmente de forma modular.

Protocolos e interfaces Una plataforma de IoT es una pasarela de conexión entre dispositivos físicos y software en la nube. Por lo tanto debe de ser capaz de coordinar, orquestar y comunicarse con cuantos más dispositivos y protocolos distintos mejor.

Independencia con el hardware Las redes IoT son heterogéneas por definición. Pueden estar formadas por sensores, ordenadores o incluso programas informáticos. Por lo tanto, el sistema operativo y el software de los dispositivos deben tener una gran compatibilidad y poder conectarse con la plataforma independientemente del hardware donde se ejecute.

Independencia con la nube Los elementos de la plataforma encargados de la recolección, monitorización y gestión de los datos pueden alojarse *on-premise* o en algún proveedor *cloud*, como por ejemplo Amazon Web Services (AWS) o Google Cloud Platform (GCP). La solución deberá diseñarse para ambos casos y así poder evitar inconvenientes a futuro, por ejemplo si hiciera falta pasar de alojarlo en infraestructura propia a la nube o si se quisiera migrar de un proveedor a otro.

Arquitectura y tecnologías Las piezas de software que formen la plataforma deberán escogerse de tal manera que sean flexibles, para poder modificarlas, y que en un periodo a corto o medio plazo no se queden obsoletas.

Seguridad Se suele decir que 'la S en IoT significa seguridad'(Tamboli, 2019). A la hora de diseñar una plataforma de este tipo existen más inconvenientes que en sistemas convencionales en cuanto a la ciberseguridad. Por ejemplo, el usuario tiene acceso físico al hardware, los dispositivos son de multitud de proveedores y muchos de ellos no pasan auditorías de seguridad, suelen utilizar Linux lo cual permite compilar malware muy fácilmente, etc. Esto ha dado lugar a la creación de diferentes *botnets* que suponen un problema a nivel mundial(Dickson, 2020).

Soporte Ya sea con el propósito de ofrecer integración con usuarios externos o para que fabricantes puedan conectar sus dispositivos a la plataforma sin necesidad de ayuda, habrá que desarrollar una documentación completa de todos los apartados de la solución.

2.2.4. Estado actual de las plataformas IoT

Actualmente existen más de 500 plataformas IoT(Tamboli, 2019), cada una con una arquitectura distinta. Algunos ejemplos de las plataformas de código libre más conocidas son Eclipse Hono™(Eclipse-Foundation, 2019) o ThingsBoard(ThingsBoard, 2021). Ambas permiten recoger mensajes de diversos protocolos de comunicación (HTTP, MQTT, CoAP, propietarios, etc.), encriptan la comunicación mediante TLS (*Transport Layer Security*), proveen de una API para disponer de los datos desde servicios externos y permiten crear reglas que ejecuten operaciones en función de los eventos recibidos, entre otras funcionalidades. ThingsBoard además permite crear *dashboards* personalizados y tiene una interfaz de creación de reglas más amigable basada en bloques en lugar de código.

Los grandes proveedores de servicios en la nube también tienen soluciones para IoT, aunque unas sean más genéricas que otras.

Google Cloud Dispone de un componente llamado *Cloud IoT Core* que permite la recogida y monitorización de datos de dispositivos IoT. El resto de tareas como la creación de *dashboards* o el procesamiento y enriquecimiento de los datos quedan de la mano de los desarrolladores utilizando el resto de componentes del proveedor: Pub/Sub, Dataflow, Data Studio, Data Lab, etc.

Amazon Web Services Propone algo similar, aunque con un mayor número de componentes dedicadas al IoT. Por un lado, disponen de un sistema operativo para microcontroladores llamado FreeRTOS, que facilita la comunicación con AWS IoT Core, el equivalente a Google Cloud IoT Core. Se utilizaría AWS IoT Device Defender para la seguridad junto con AWS IoT Device Management para la gestión de dispositivos.

Soluciones más completas serían AWS IoT Analytics para análisis sofisticados de los datos recogidos, AWS IoT SiteWise para recolección y análisis a gran escala, AWS IoT Events para creación de reglas o AWS IoT Things Graph para crear flujos y aplicaciones.

Microsoft Azure También dispone de una colección de servicios enfocados al IoT. Al igual que AWS tiene un sistema operativo para los dispositivos: Windows IoT, y una herramienta de recolección y monitorización de eventos: Azure IoT Hub, con encriptado

incluido. Azure Digital Twins permite la creación de aplicaciones y otros servicios como Azure IoT Edge o Azure Time Series Insights amplían las funcionalidades que se pueden conseguir.

Por supuesto, otros proveedores cloud tienen sus alternativas. Por ejemplo, IBM ofrece su producto Watson IoT Platform[1] o el gigante asiático Alibaba su Alibaba Cloud IoT Platform[2].

Por otro lado, Guth et al. (2018) comparan cuatro plataformas de código libre y cuatro propietarias en cuanto a su arquitectura. Las libres son FIWARE[3], OpenMTC[4], SiteWhere[5] y Webinos[6]. Las propietarias son AWS IoT, IBM Watson IoT, Azure IoT Hub y Samsung SmartThings[7]. En este mismo artículo los autores presentan una arquitectura de referencia basada en cuatro capas: dispositivos, *gateway* (enlace entre los dispositivos y el resto de componentes de la plataforma), *middleware* para integración y aplicación. Las ocho plataformas cumplen esta definición en cuatro capas, cada una con una menor o mayor complejidad. Algunas de ellas utilizan el concepto de dispositivos 'inteligentes', por lo que incluyen algún tipo de reglas o funcionalidad lógica. Todas soportan tanto sensores como actuadores. En la primera tabla del artículo se recoge una comparativa muy completa de los componentes cada plataforma agrupados en esas cuatro capas.

Otro proyecto interesante es el propuesto en Pastor-Vargas et al. (2020). En él se define un entorno educativo para que los estudiantes desarrollen proyectos de IoT. Utilizan un despliegue en contenedores Docker orquestado por Kubernetes sobre un clúster de Raspberry Pi. Los dispositivos IoT son sensores que se conectan a estas tarjetas, como por ejemplo cámaras o micrófonos. La conexión entre los dispositivos y la plataforma se realiza mediante el protocolo de mensajería MQTT.

Aunque no sea un ejemplo de una plataforma IoT comercial como las anteriores, sirve para entender cómo otros equipos han desarrollado pruebas de concepto con hardware de bajo coste. Además, en el artículo incluyen ejemplos de prácticas que han propuesto a

[1] https://www.ibm.com/internet-of-things/
[2] https://www.alibabacloud.com/product/iot
[3] https://www.fiware.org/
[4] https://www.open-mtc.org/ (comparte parte de los componentes con FIWARE)
[5] https://www.sitewhere.org/
[6] https://www.webinos.org/ (descontinuada)
[7] https://www.samsung.com/es/apps/smartthings/

los estudiantes con ideas muy interesantes, como utilizar los servicios en la nube de IBM para almacenamiento externo de los datos.

2.2.5. ¿Crear o contratar?

Si existen todas estas alternativas, tanto libres como privadas, ¿qué necesidad hay de diseñar una propia? Existen tres opciones distintas a la hora de diseñar una plataforma IoT:

- Contratar una de las soluciones que ofrecen una gestión completa e integrarla con los servicios que se vayan a utilizar.
- Contratar los componentes cloud necesarios y construir todo sobre ellos, como proponen Google, AWS o Azure.
- Construir una propia desde cero, con la posibilidad de alojar la solución en la nube.

El primer punto a favor a la hora de construir una plataforma desde cero es que permite focalizar el esfuerzo en la dirección que se quiera tomar, sin necesidad de modificar el enfoque hacia lo que los proveedores ofrecen. Por ejemplo, si el caso de uso es muy concreto y no se necesita todo lo que las soluciones cloud proveen, crear una plataforma con lo justo y necesario para su cometido suele ser la mejor opción.

Por otro lado, podría parecer que alojar las máquinas necesarias puede suponer un alto coste a la hora de la adquisición de estos equipos y contratar a personal para gestionarlo, pero siempre queda la opción de desplegar la solución en algún proveedor cloud. Que sea desarrollo propio no excluye que se pueda usar IaaS para el alojamiento o productos como puedan ser Cloud Pub/Sub o AWS EMR para tareas concretas o pruebas de concepto.

Una ventaja de las plataformas ya construidas es que permiten reducir el llamado *time to market* (TTM), es decir el tiempo que lleva alcanzar un producto mínimo viable. Contratar una de estas soluciones podría reducir a su vez costes a corto plazo, pero a la larga podría tener el efecto contrario. Algunos proveedores establecen un precio por dispositivo o usuario y otros por tráfico. La previsión a medio o largo plazo es algo que se debe tener en cuenta a la hora de tomar la decisión entre crear una plataforma desde cero o contratarla. Otra opción es la licencia de por vida que empresas como ThingsBoard

ofrece, que no supondría gastos por un aumento de dispositivos pero requiere alojamiento y escalado por parte del cliente cuando fuera necesario.

Sin embargo, si por el motivo que sea se elige contratar una de estas plataformas para reducir el TTM, habrá que tener en cuenta que cuanto más difiera la definición del producto que se quiere construir de lo que oferta el proveedor, más modificaciones habrá que hacer. Más aún si se prevé mantener el servicio durante años e ir añadiendo funcionalidades o integraciones. Sería más óptimo retrasar el TTM pero reducir el esfuerzo que lleva añadir nuevos componentes por ser código propio y no de terceros.

Por último, una de las ventajas más importantes de construir una versión propia es la velocidad con la que se puede innovar. Si una nueva tecnología irrumpe en el campo del IoT o del Big Data, se debe esperar el tiempo necesario para que los proveedores cloud la incorporen a sus servicios y después dedicar tiempo y esfuerzo en adaptar los cambios en la aplicación que se haya desarrollado. Por ejemplo, cada vez que se publica una nueva versión de Apache Kafka, que se detalla más adelante, los proveedores tardarán un tiempo en actualizar sus productos. Un caso concreto sería la versión 2.6.0 de Kafka que se publicó el 3 de agosto de 2020. Amazon MSK, el servicio de AWS para provisión de clústeres Kafka actualizó la versión el 21 de octubre de ese mismo año: 2 meses y medio más tarde.

2.3. ¿Qué es el Big Data?

El término hace referencia a colecciones de datos lo suficientemente grandes y complejos como para requerir un procesamiento no tradicional, como podría ser la computación distribuida. Tiene seis características principales llamadas 'las 6 Vs del Big Data' (Ristevski and Chen, 2018), aunque el número varía según la fuente que se consulte:

- **Volumen**: se trabaja con terabytes o incluso petabytes de datos.
- **Velocidad**: se generan datos con alta frecuencia y, en muchos casos, deben ser procesados en tiempo real.
- **Variedad**: las fuentes de datos y los tipos pueden ser muy diversos. Se trabaja con texto plano, imágenes, audio, vídeos...

- **Veracidad**: se pueden recibir datos estructurados o no estructurados con ruido o alteraciones además de poder corromperse en la transmisión.

- **Valor**: los datos deben ser útiles para el análisis que se hace en tiempo real o una vez almacenados. Además tienen cierta volatilidad: los datos son válidos durante un periodo de tiempo determinado.

- **Variabilidad**: la dimensionalidad, tipos, inconsistencias, y la velocidad de generación de los datos puede cambiar con el tiempo.

Una de las tecnologías que comparte estas seis características con el Big Data es el IoT. Cualquier plataforma de IoT que busque administrar un gran número de dispositivos o un número más reducido pero con una tasa de recolección de datos muy elevada, necesitará de las herramientas de recolección, procesamiento y visualización de datos utilizadas en el Big Data.

2.4. ¿Cómo se procesa el Big Data?

Por lo general en Big Data se trabaja con procesos ETL (del inglés *Extract, Transfor and Load*) que constan de tres etapas:

- **Extracción**: los datos deben extraerse de las fuentes de datos origen, que pueden ser bases de datos, ficheros o dispositivos IoT, entre otros. Una tecnología ampliamente utilizada en esta etapa son los *frameworks* de procesamiento de *streams*. Destaca Apache Kafka (Kreps et al., 2011) como solución de código libre y Confluent, Google Cloud Pub/Sub y AWS Kinesis como alternativas en la nube.

- **Transformación**: los datos deben ser tratados previo a su almacenamiento. En esta etapa se modifica el formato de los datos serializándolos en formato Json, Avro o Parquet, por ejemplo, se enriquecen con otras fuentes de datos, se reduce el número de campos para ahorrar en almacenamiento, etc. Una herramienta esencial para esta etapa sería Apache Spark (Zaharia et al., 2010) o Google Cloud Dataflow, si se prefiere en la nube.

- **Carga**: los datos se almacenan en bases de datos o en un *data warehouse*. Tanto Apache Kafka como Apache Spark, y sus alternativas en la nube, tienen conectores para el guardado de los datos en las bases de datos más conocidas (tanto relacionales como no relacionales (Phiri and Kunda, 2017)) o en *data warehouse* tales como Hadoop (Polato et al., 2014), AWS S3 o Cloud Storage, entre muchos otros.

Por supuesto, existen soluciones cloud que engloban las tres etapas, como por ejemplo AWS Glue.

2.4.1. Arquitecturas más comunes

El análisis, la monitorización y la visualización de datos se puede realizar tanto en tiempo real como en *batch*, es decir por lotes o conjuntos de datos. La mayoría de situaciones se solventan o bien con una arquitectura Lambda o con una Kappa (Singh et al., 2019), en las Figuras 2.1 y 2.2. Otras posibles arquitecturas se derivan de estas dos, añadiendo o suprimiendo componentes.

La **arquitectura Lambda** se divide en dos capas: una *streaming* y otra *batch* (Figura 2.1). La primera se encarga de responder ante consultas sobre los últimos datos y la segunda de almacenar en bruto y en tablas agregadas o indexadas, los datos procesados por lotes. El resultado de las consultas será una combinación de ambas. La ventaja es que la capa *batch* permite un análisis en retrospectiva del total de los datos recibidos además de reducir la latencia de consultas. Una desventaja es que se duplica el sistema de procesamiento al utilizar dos capas en lugar de solamente una.

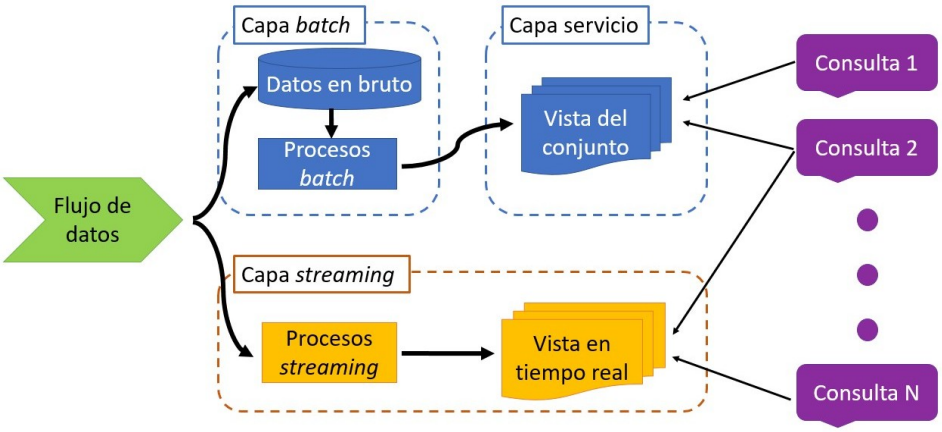


Figura 2.1: Arquitectura Lambda.

La **arquitectura Kappa** está formada solamente por la capa *streaming* (Figura 2.2), que debe responder también ante consultas sobre datos históricos. Mientras que se reduce la complejidad del sistema, aumenta la latencia de consultas históricas y se pierde retrospectiva. Además, el almacenamiento de datos estará más limitado a no ser que se vuelquen los mensajes a un *data warehouse*.

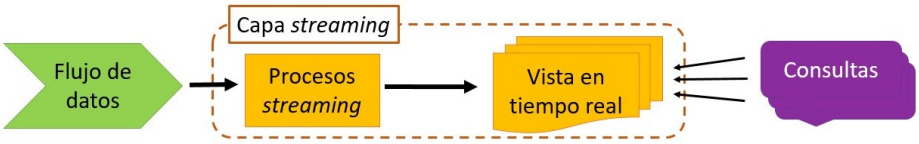
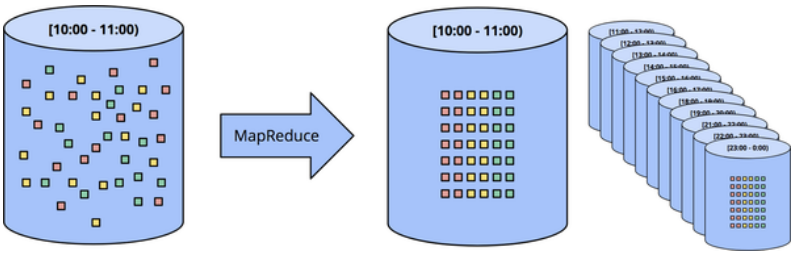


Figura 2.2: Arquitectura Kappa.

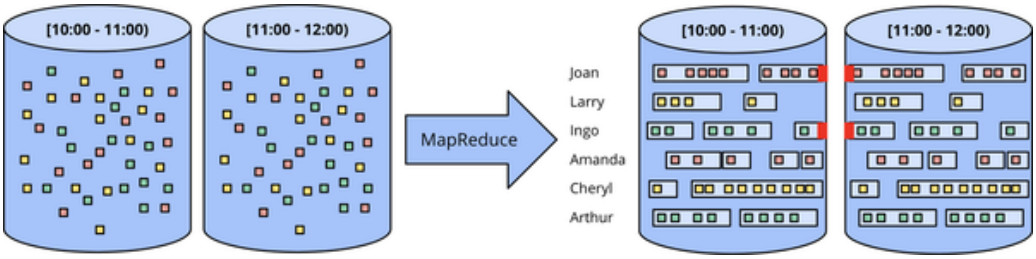
2.4.2. Diferencias entre el procesamiento batch y en streaming

El tratamiento en *batch* se suele realizar mediante ventanas fijas o sesiones (Tyler Akidau, 2018), dependiendo de la situación. Un esquema de estas soluciones se muestra en la Figura 2.3. Las primeras tratan de abordar el procesamiento de datos masivos separando el conjunto inicial en ventanas de un tamaño fijo e iterando hasta procesar el total de los elementos. En la Figura 2.3a la duración de las ventanas es de una hora y el procesamiento que se hace es simplemente una tarea MapReduce (Dean and Ghemawat, 2004) que ordena los datos. En el *data warehouse*, que sería lo que se representa como una fila de silos en la parte derecha, se almacenan los datos separados en esas ventanas.

Por otro lado, las sesiones dividen el procesamiento aún más. Una sesión se define como un periodo de actividad que finaliza con un periodo de inactividad. Un caso muy común es la analítica web, donde se estudia la interacción de un usuario con la página desde el momento en el que accede hasta que la abandona. La Figura 2.3b podría ser un ejemplo de este caso de uso. Se aplica de nuevo una tarea MapReduce que ordena los datos en función del usuario que los genera. En este caso se utilizan *batches* de una hora que pueden compartir sesiones, es decir que algunas pueden quedar interrumpidas entre *batches* consecutivos. Este es el caso de los usuarios Joan e Ingo, tal y como se remarca en rojo sobre la imagen. Se puede reducir la frecuencia de estos cortes aumentando el tamaño del *batch*, a cambio de aumentar la latencia ya que cuantos más datos entren en cada *batch* mayor será el tiempo de procesamiento de éstos. Otra opción sería añadir una



(a) Ventanas fijas.



(b) Sesiones.

Figura 2.3: Soluciones al procesamiento de datos masivos en *batch*. Referencia: Tyler Akidau (2018).

lógica adicional que acumulara los datos la sesión en *batches* previos a la actual y así poder tratarla como una completa.

El tratamiento de datos en *streaming*, al contrario que en *batch*, debe ser capaz de procesar datos desordenados en el tiempo y deben asumir que puede que se haya perdido algún mensaje. Para ello se pueden utilizar varios enfoques dependiendo de la situación (Tyler Akidau, 2018):

Tiempo irrelevante (Time-Agnostic) Siempre y cuando se pueda hacer el tratamiento de los datos según se cargan en el sistema, sin tener en cuenta el ordenen en el que llegan los mensajes, se podrán utilizar las técnicas *time-agnostic* de filtrado y unión interna. Existen más, pero estas son las más comunes:

Filtrado Este es uno de los métodos más sencillo cuando el tiempo es irrelevante. Consiste aplicar filtros para reducir la cantidad de información que atraviesa el sistema. Un caso de uso podría ser el análisis del tráfico de un conjunto de páginas web en tiempo real. Si solamente se quiere monitorizar una, se filtra el total de los datos manteniendo solo aquellos que interesen. Por ejemplo, en la Figura 2.4 se seleccionan solamente los eventos de color amarillo.

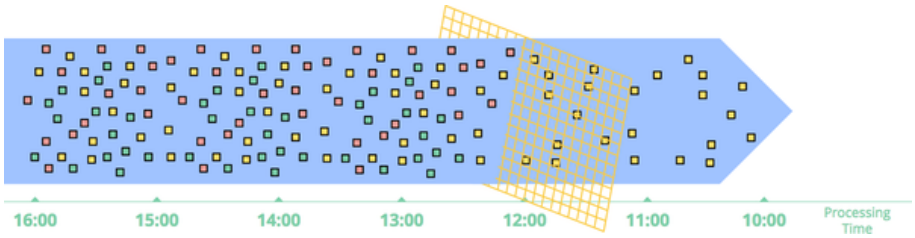


Figura 2.4: Filtrado de un flujo de datos en tiempo real. Referencia: Tyler Akidau (2018).

Unión interna (Inner joins) Si lo que se busca es combinar elementos de varios flujos cuando estos lleguen, no importa el tiempo. El sistema guardará los datos en un buffer hasta que llegue el último elemento necesario para la unión. En la figura 2.5 se espera a que un dato de tipo cuadrado y uno circular de un mismo color lleguen para unirlos y enviarlos al siguiente paso de la *pipeline*. Habrá que establecer por supuesto algún sistema de 'recolección de basura' o un *timeout* para no acumular eventos en caso de que algún dato se pierda.

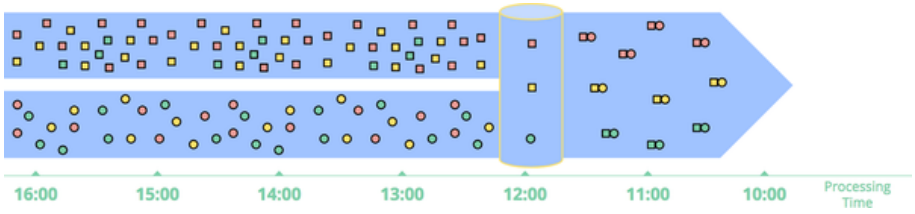


Figura 2.5: *Inner join* de dos flujos de datos en tiempo real. Referencia: Tyler Akidau (2018).

Algoritmos de aproximación Se pueden utilizar modelos que dados unos datos de entrada generen una salida que se asemeje a lo que cabría esperar. En este caso el tiempo puede llegar a ser relevante ya que cuantos más datos se tengan más realista será la aproximación que haga el algoritmo.

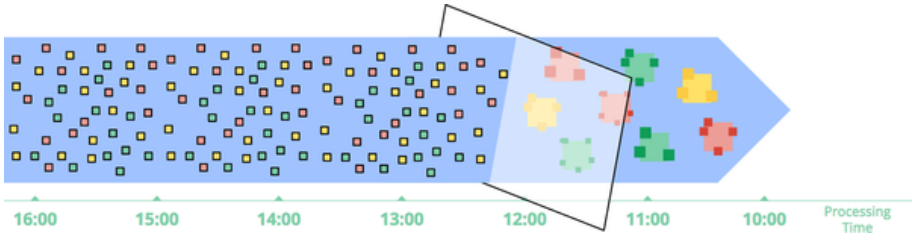


Figura 2.6: Algoritmo de aproximación aplicado a un flujo de datos en tiempo real. Referencia: Tyler Akidau (2018).

Ventanas (Windowing) Al igual que en *batch*, en *streaming* también se pueden usar ventanas(Gerard Maas, 2019). El flujo de datos se puede dividir de tres maneras distintas, siendo el tiempo y el orden en el que llegan los datos muy importante en cada una de ellas. En la Figura 2.7 se muestran estos tres casos y se detallan a continuación.

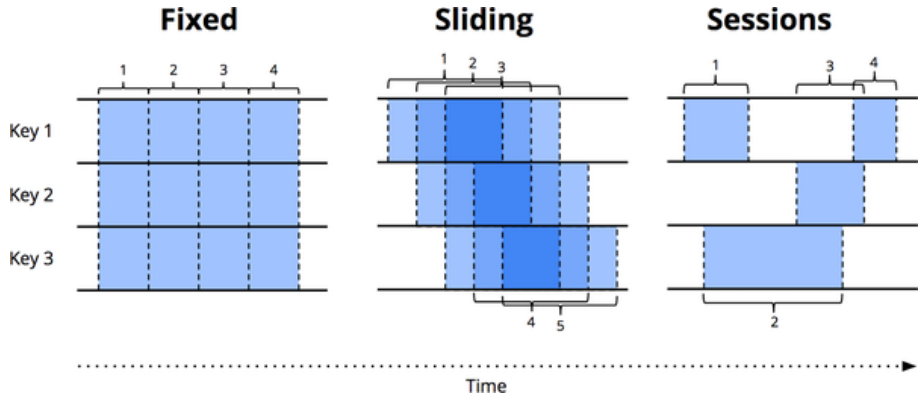


Figura 2.7: Procesamiento en ventanas fijas, deslizantes y sesiones de un flujo de datos en tiempo real. Referencia: Tyler Akidau (2018).

Ventanas fijas El *stream* de datos se divide en segmentos de duración fija. En algunos casos se puede separar por alguna clave, como en la Figura 2.7, pero no es necesario. Esto puede agilizar el procesamiento de los datos ya que permite balancear la carga en distintas máquinas.

Ventanas deslizantes (Sliding windows) Al igual que en el caso anterior las ventanas son de tamaño fijo pero se añade otra variable que es el *periodo*. Si el tamaño de las ventanas y el periodo es el mismo, es equivalente a trabajar con ventanas fijas. Sin embargo, si el periodo es menor, las ventanas se solapan permitiendo un análisis más realista de los datos según alcanzan el sistema. Por otro lado, si el periodo es mayor que el tamaño de la ventana se estarían tomando muestras periódicas de partes del flujo de datos, sin analizarse la totalidad de los mensajes.

Sesiones Al igual que en *batch*, las sesiones estarían formadas por secuencias de eventos de un mismo tipo y separadas por un periodo de inactividad. El tamaño de la ventana de sesión no puede definirse *a priori*.

Se ha hablado de la relevancia del tiempo en el procesamiento del *stream*, pero se puede

trabajar con dos conceptos distintos(Traub et al., 2018): el **tiempo de procesamiento** es el instante en que el dato alcanza el sistema de recolección de datos y se aplica la transformación pertinente. Por otro lado, el **tiempo de evento** es el momento en que el dato es generado. Se pueden usar ambos indistintamente para la división en ventanas pero cada uno tiene sus ventajas y sus inconvenientes.

Utilizar el tiempo de procesamiento es muy sencillo, ya que no es necesario comprobar que los datos estén ordenados o completos, es decir que hayan llegado todos los necesarios. Es ideal en situaciones donde se quiera extraer información según es observada, como por ejemplo para predecir caídas de servicios web. No importa cuándo se generan las peticiones sino cuándo sean gestionadas y si ha habido un decremento brusco.

Ahora bien, si se quieren analizar los datos según se generan será necesario utilizar el tiempo de evento. La mayoría de *frameworks* de procesamiento en *streaming* (Kafka, Flink, Spark, Hadoop...) soportan esta gestión de forma nativa. Como se observa en la Figura 2.8, cuando se dividen los datos utilizando el tiempo de procesamiento pueden quedar datos fuera de estas ventanas (aquellos que se señalan con flechas) y el análisis ser incorrecto. Los motivos pueden ser diversos: una caída en la red, saturación del sensor, etc. Para evitar estas situaciones es necesario guardar los eventos en *buffers* u otras técnicas (Traub et al., 2018) como por ejemplo las *marcas de agua* (*watermarks*) para determinar si un dato entra dentro de la ventana o se considera un dato tardío (*late data*).

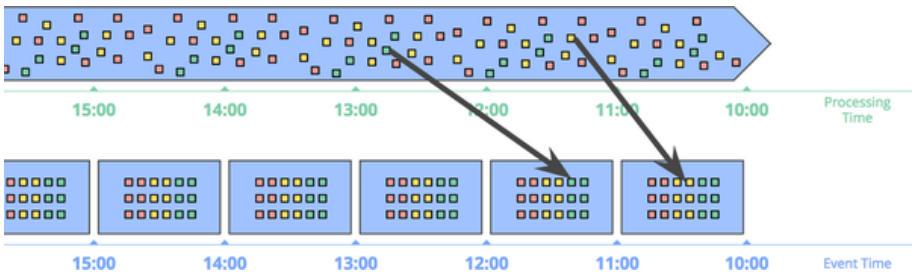


Figura 2.8: Procesamiento en ventanas fijas de un flujo de datos en tiempo real utilizando el tiempo de evento. Referencia: Tyler Akidau (2018).

2.5. Resumen del capítulo

La sección ha servido para introducir los conceptos con los que se va a trabajar en la implementación de la plataforma. Son importantes sobretodo las arquitecturas y las

técnicas de procesamiento de datos, ya que serán piezas clave a la hora de seleccionar las tecnologías. No menos importante es la justificación del porqué merece la pena diseñar una plataforma desde cero en lugar de partir de una comercial. Sin ella el objetivo del trabajo pierde sentido.

Capítulo 3

Diseño de la arquitectura

3.1. Introducción

En esta sección se definen los componentes esenciales que deben estar presentes en cualquier plataforma IoT. Se mencionan además otros que suelen utilizarse pero que no son obligatorios, como los microservicios o las integraciones con otras aplicaciones. Se comprueba también que no existe una definición cerrada de cómo ha de ser una plataforma ni de las funciones de sus componentes, ya que en algunos casos unos se engloban dentro de otros.

A continuación se propone la arquitectura de la prueba de concepto que se desarrolla en este trabajo, formada por supuesto por los componentes fundamentales de las plataformas. Se le añade además uno de monitorización para poder visualizar el estado de ésta y del *hardware* sobre el que se instale.

3.2. Arquitectura general

No existe una definición cerrada de cómo ha de ser la arquitectura de una plataforma IoT. Además pueden ser tanto de tipo Lambda como Kappa, arquitecturas que engloban la gran mayoría de soluciones pero que no concretan qué elementos ni herramientas usar.

Por ello, la definición de arquitectura deberá basarse en unos componentes esenciales y otros que aunque aporten gran valor son opcionales. A partir de Tamboli (2019) y las

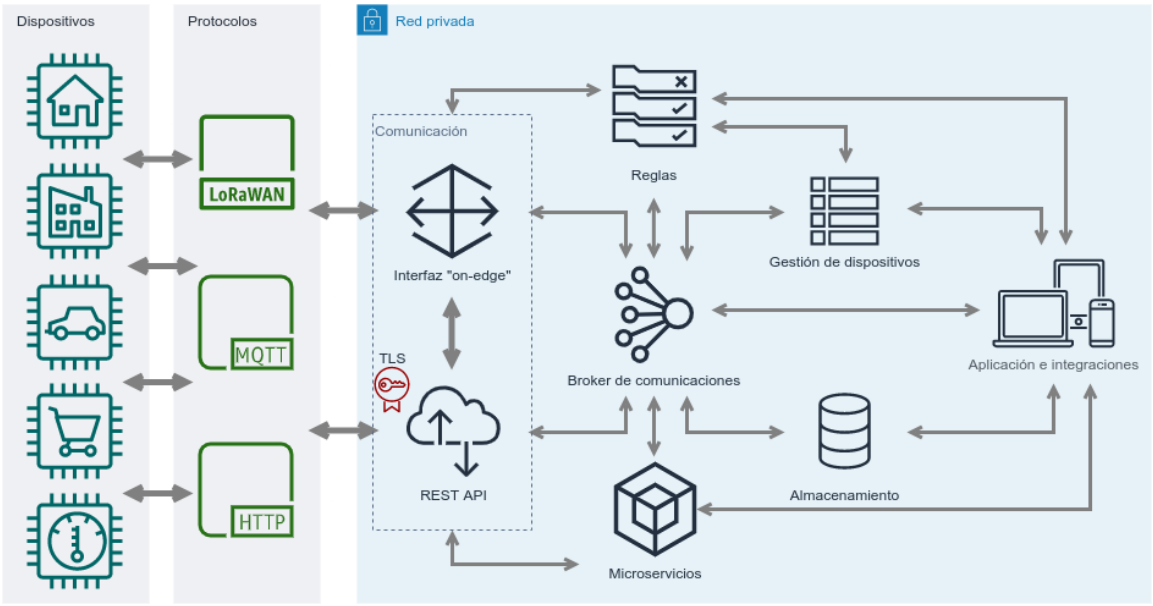


Figura 3.1: Arquitectura general de una plataforma IoT.

arquitecturas comerciales presentadas en la sección anterior, se podría concluir que los elementos que ha de tener una plataforma IoT son los que se muestran en la Figura 3.1 y defino a continuación:

Dispositivos La fuente de datos es un componente esencial y no tendría sentido hablar de una plataforma IoT sin dispositivos -sensores o actuadores- que ingesten dato en ella.

Comunicación o *Gateway* Las *gateways* son el componente que establece la comunicación entre la plataforma IoT y los dispositivos. Muchos dispositivos tienen acceso a Internet y podrán utilizar protocolos como HTTP o MQTT para enviar información. Sin embargo, otros pueden utilizar comunicación por radiofrecuencia -con protocolos como LoRa (Augustin et al., 2016)-, Bluetooth, redes móviles, etcétera.

Para simplificar la conexión con la plataforma se crean *gateways* para tantos protocolos como sea necesario, aunque en gran parte de casos servirá con HTTP y HTTPS. De esta forma se abstrae el protocolo de comunicación del resto de componentes. Además, estas *gateways* pueden ser tan complejas como se requiera e incluir una primera etapa de tratamiento del dato previo a la ingesta, autenticación, eliminación de ruido o balanceo de carga, entre otras funciones. Para ello puede ayudarse de otros componentes, como son el de reglas o los microservicios.

Este componente **no solamente se utiliza para recoger dato**. Puede que los dispositivos necesiten consultar algún tipo de información disponible en la plataforma, por ejemplo la frecuencia con la que deben tomar dato, o utilizar microservicios disponibles en la red interna de la plataforma.

En la Figura 3.1 se ha representado tanto una API REST para comunicación HTTP, como una interfaz *on-edge* para dispositivos que no dispongan de conexión a Internet.

***Broker* de comunicaciones** Un *broker* es un intermediario entre un publicador y un suscriptor. En una plataforma IoT es el encargado de recibir el dato en *stream* o *batch* y distribuirlo entre el resto de componentes, ya sea mediante PUSH o PULL. Al igual que el *gateway* puede utilizar otros componentes para orquestar exportaciones de los datos y decidir cómo y dónde se van a guardar, enriquecerlos, etcétera.

Las herramientas más utilizadas como *broker* de comunicaciones son Apache Kafka y RabbitMQ[1].

Almacenamiento En la gran mayoría de los casos, el dato que se recibe se almacena para poder utilizarlo más adelante, principalmente para análisis. Sin embargo, pueden existir aplicaciones donde la plataforma solo se utilice para iniciar eventos o lanzar alarmas en función de la información que llegue y no sea necesario almacenarlo.

Se pueden utilizar tanto bases de datos SQL (*Structured Query Language*) como NoSQL (*Not only SQL*), o incluso sistemas de archivos como Hadoop. Algunas de las mejores opciones para plataformas IoT con capacidad de procesamiento para Big Data serían HiveQL o BigQuery, para dato relacional, y Cassandra o MongoDB para no relacional.

Otros componentes Algunos de los componentes que complementan los anteriores y que se pueden ver en la mayoría de plataformas IoT son los siguientes:

Microservicios Permiten reutilizar código y modelos probabilísticos o de inteligencia artificial entre los distintos componentes. Son de gran utilidad para dispositivos con

[1] https://www.rabbitmq.com/

baja capacidad computacional, ya que pueden enviar sus medidas a estos microservicios y obtener las predicciones necesarias.

Lo más común es que estos microservicios se empaqueten en contenedores, que son similares a las máquinas virtuales pero comparten un mayor número de propiedades con el sistema operativo en el que se alojan y tienden a ser más ligeros. La principal característica es que son independientes del sistema en el que se instalen, así que se pueden distribuir de forma sencilla. La herramienta más utilizada para contenerización es Docker[2].

Módulo de reglas El módulo de reglas puede tener varias funciones: se puede utilizar para orquestar servicios en función de qué mensajes se reciban, cómo actuar cuando un nuevo dispositivo trata de registrarse en la plataforma o crear alertas, por ejemplo. Podría usarse también para enriquecer el dato entrante o para enrutar los mensajes en función del contenido y quien lo envíe.

Se podrían utilizar varias herramientas para este módulo. Thingsboard, una de las plataformas que se presentaron en la sección anterior, tiene un módulo de reglas muy completo y sencillo de utilizar que hace uso de una interfaz gráfica y Javascript para definir las reglas[3].

Una solución más sencilla, por ejemplo para autenticación o enrutamiento, podría ser una base de datos en memoria como Redis, que es sencilla de implementar y provee de un acceso de lectura y escritura muy rápido.

Gestión de dispositivos Este módulo podría incluirse tanto en el de almacenamiento y como en el de reglas dependiendo de su función. En caso de utilizarse como un registro de los dispositivos que se han conectado a la red y de estadísticas que incluyan configuraciones o cantidad de mensajes enviados, podría incluirse en el primero. Si se utiliza como una interfaz para gestionar qué dispositivos pueden publicar mensajes, límites de tamaño de los mensajes o frecuencia de publicación, por ejemplo, entraría dentro del módulo de reglas.

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

[3] https://thingsboard.io/docs/user-guide/rule-engine-2-0/re-getting-started/

Aplicaciones e integraciones Aunque no todas las plataformas dispongan de él, este es uno de los módulos más importantes por el gran valor que puede llegar a aportar. No es obligatorio ya que con almacenar los datos serviría para considerar una plataforma IoT como tal. Sin embargo, disponer de una interfaz de monitorización o de análisis de datos puede ser realmente ventajoso.

Ejemplos de aplicaciones podrían ser Grafana y Prometheus para monitorización. Las integraciones incluirían tanto desarrollos de terceros como herramientas públicas como Apache Spark, que pudieran conectarse al almacenamiento o al *broker* directamente y consumir o publicar datos.

3.3. Arquitectura propuesta

Para este trabajo se van a implementar los módulos fundamentales de las plataformas IoT. Sin embargo, en la sección 3.4 se proponen varias arquitecturas que utilizan un mayor número de componentes.

En la Figura 3.2 se puede ver la propuesta. Para simular los dispositivos se ha desarrollado un código en Go que envía peticiones a la API REST conteniendo un campo para identificación y otro con el dato que se quiere almacenar. También es posible publicar utilizando `cURL`[4] u otra librería de peticiones HTTP. Se ha pensado así para que cualquier dispositivo con acceso a Internet pueda utilizar la plataforma, sin necesidad de instalar ningún conector o desarrollar un programa específico para ello. Se incluye además un módulo de reglas que utilizará esta API REST para comprobar que el dispositivo y el mensaje cumplan los requisitos establecidos por los administradores de la plataforma.

Las APIs REST se han desarrollado también en Go. La primera se conecta a un clúster de Apache Kafka, que actúa tanto de *broker* como de almacenamiento, y la segunda a un clúster de Redis. Para monitorizar el estado de los componentes se ha utilizado Prometheus y Grafana, definidos en la siguiente sección. El primero se encargará de recoger las métricas de cada recurso del clúster y el segundo de representarlas en un *dashboard*.

Por supuesto, tanto el almacenamiento en memoria de Redis como el clúster de Kafka pueden usarse por integraciones que se hagan a futuro para añadir nuevas funcionalidades

[4] https://curl.se/

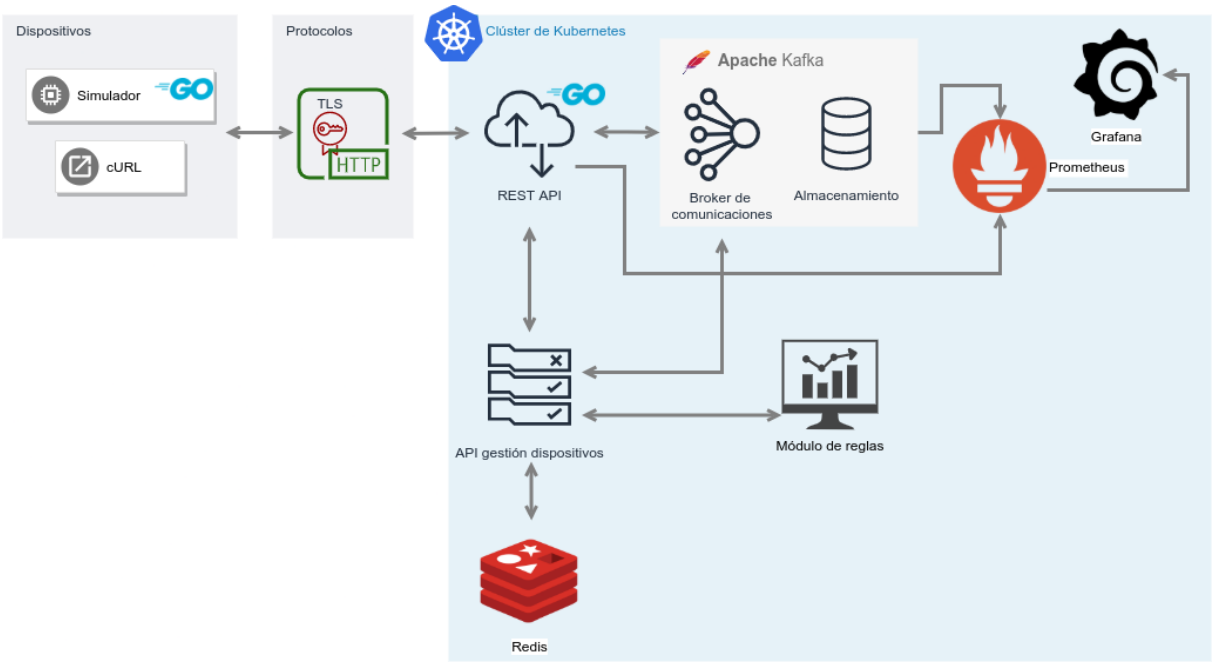


Figura 3.2: Arquitectura propuesta para la prueba de concepto.

a la plataforma.

Con el fin de que la plataforma sea replicable y agnóstica en cuanto al proveedor cloud o sistema en el que se despliegue, se ha utilizado Kubernetes para gestionar el conjunto de componentes y Helm para su distribución entre clústeres.

3.4. Casos de uso

El desarrollo de esta plataforma IoT no cierra puertas a ningún área. Se podría aplicar en agricultura, ciberseguridad, marketing, manufactura o desarrollo web, por ejemplo.

Un producto que ha ganado gran popularidad entre las empresas de marketing en los últimos años son los CDP (*Customer Data Platform*). Plataformas que integran dato de cliente, ya sean transacciones o dato personal, para fuentes heterogéneas con el objetivo de tener una visión completa de la interacción del usuario con la empresa. Una de las más conocidas es AEP[5] (Adobe Experience Platform), que permite definir un esquema con los campos disponibles para los usuarios, ingestar dato desde multitud de fuentes e integrarlo para rellenar la mayor cantidad de campos del esquema como sea posible. Para

[5] https://business.adobe.com/es/products/experience-platform/adobe-experience-platform.html

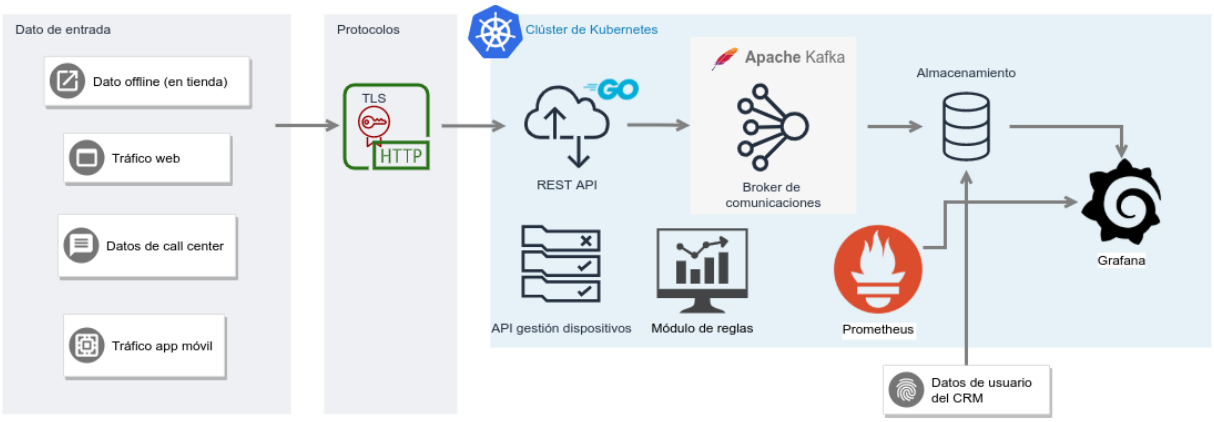


Figura 3.3: Caso de uso de la plataform IoT como un CDP.

ello utiliza un *broker*, un *data lake* para almacenamiento y un módulo de reglas basado en OpenWhisk, entre otras muchas herramientas.

Esta arquitectura es muy similar a la que se ha propuesto en este trabajo, por lo que esta plataforma se podría usar para desarrollar un CDP. Un posible diseño se muestra en la Figura 3.3.

En el campo de la ciberseguridad se podría utilizar para monitorizar una *botnet* por ejemplo. Otra forma de utilizarla sería para evitar que a una red de dispositivos IoT se les instale *malware*. Se podrían configurar para enviar un *hash* del código fuente a la plataforma cada cierto tiempo y así poder ver si se ha visto modificado o incluso enviar los logs del sistema operativo y monitorizarlos en conjunto desde la plataforma. En agricultura, manufactura o industria el caso de uso más fácil de imaginar es para monitorización de maquinaria, sensores y otros dispositivos conectados a Internet.

3.5. Comparación con otras plataformas

Se pueden estudiar las diferencias entre el modelo propuesto y las plataformas que se analizan en Guth et al. (2018) en función de sus capacidades y arquitecturas.

3.5.1. Capacidades de las plataformas

En la tabla 3.1 se recoge una comparativa de las funcionalidades de las distintas plataformas junto con la propuesta en este trabajo. El asterisco en la fila de monitorización

indica que la plataforma tiene una herramienta disponible para que el desarrollador cree el dashboard, pero no se incluye por defecto. Las **desventajas *cloud*** son un alto coste a la larga, dependencia de los servicios del proveedor (muchos no se pueden migrar a otras *cloud* o a servidores *on-premise*) y poca capacidad de modificación (gran parte de los servicios no se pueden personalizar o crear implementaciones propias). Se ha suprimido OpenMTC por compartir componentes con FIWARE y Webinos, que ya no está disponible.

Donde más diferencias hay es en el tipo de protocolos que soportan, aunque por lo general todas soportan HTTP y MQTT. Se observa que Samsung SmartThings es una plataforma claramente orientada a dispositivos inteligentes, ya que utiliza los protocolos Z-Wave y ZigBee que son los más comunes en este tipo de hardware.

Por otro lado, para el almacenamiento MongoDB es la elección de las plataformas libres mientras que las *cloud* utilizan sus propios servicios.

	FIWARE	**SiteWhere**	**AWS IoT**	**IBM Watson**	**Azure IoT Hub**	**Samsung SmartThings**	**Propia**
Sensores	✓	✓	✓	✓	✓	✓	✓
Actuadores	✓	✓	✓	✓	✓	✓	×
Gestión de dispositivos	✓	✓	✓	✓	✓	✓	×
Protocolos	HTTP, MQTT, LoRaWAN, SigFox	HTTP, MQTT, AMQP	HTTP, MQTT	MQTT, JMS	HTTP, MQTT, AMQP	Z-Wave, ZigBee	HTTP
Módulo de reglas	✓	✓	✓	Solo notificaciones	✓	✓	✓
Almacenamiento	MongoDB	MongoDB, Hbase, InfluxDB	Conectores	Conectores	Conectores	Su propio cloud	En memoria
Contenedores	✓	✓	-	-	-	-	✓
Monitorización	✓*	✓*	✓	✓	✓	✓	✓
Despliegue	Cloud o Helm	Helm	Cloud	Cloud	Cloud	Cloud	Helm
Ventajas	Comunidad y ayuda a *startups*, OpenSource	OpenSource	Integración con el resto de servicios de AWS	Extensa documentación	Integración con el resto de servicios de Azure	Especializado en dispositivos IoT domésticos	Simplicidad, extensibilidad, OpenSource
Desventajas	Complicado de modificar	Poca comunidad	Desventajas *cloud*	Desventajas *cloud*	Desventajas *cloud*	Desventajas *cloud* y orientado a hogar	Sin comunidad y componentes muy sencillos

Tabla 3.1: Comparación de los módulos de las distintas plataformas IoT que se mencionan en el trabajo.

En el caso concreto de FIWARE y SiteWhere, ambas disponen de imágenes de Docker para desplegar la plataforma e incluso ofrecen un *chart* de Helm para desplegar en un clúster de Kubernetes (repository, 2021) (SiteWhere, 2021). Sin embargo, con SiteWhere es necesario instalar el clúster de Kafka de forma manual o contratar uno en la nube.

3.5.2. Arquitecturas

Se analizan ahora desde el punto de vista de su arquitectura. Las plataformas cloud se pueden ver bien representadas por la propuesta de AWS, disponible en la Figura 3.4. Todas utilizan su *suite* de servicios cloud para complementar su plataforma IoT. Sobre esta arquitectura se pueden localizar los componentes esenciales y opcionales que se definieron en el apartado 3.2: sensores y actuadores (dispositivos), un *broker* de mensajería, registro (gestor) de dispositivos, un módulo de reglas y un grupo de aplicaciones a las que se conectaría la plataforma, lo que se denominó 'integraciones'. Al ser una propuesta de AWS, la integración con estas aplicaciones es directa y no haría falta crear los conectores correspondientes.

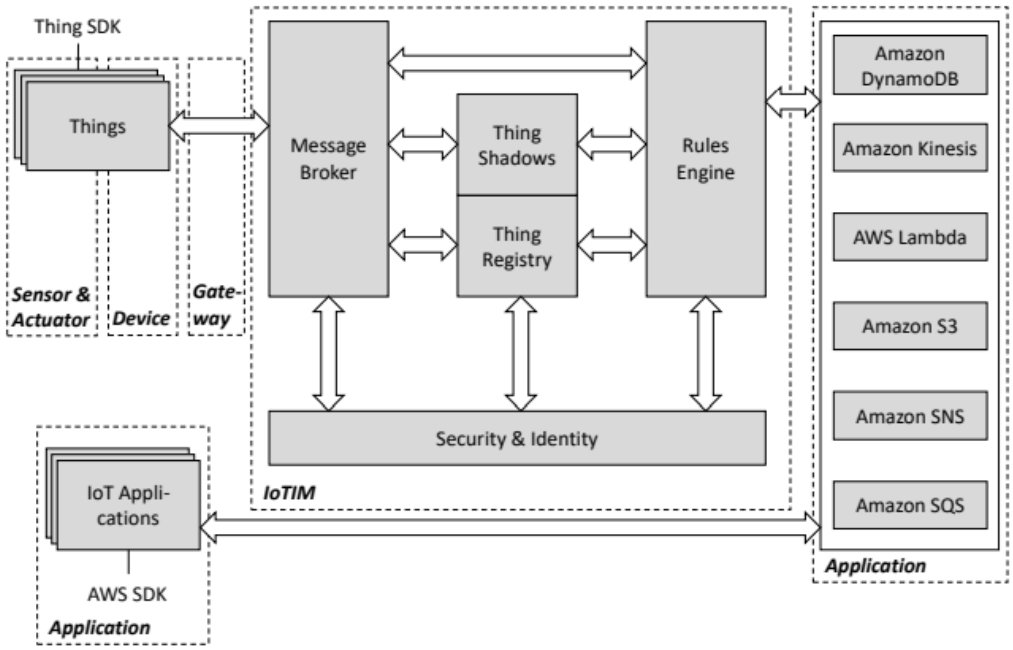


Figura 3.4: Arquitectura de AWS IoT. Fuente: Guth et al. (2018).

Realmente se podría hacer una correspondencia uno a uno entre estas aplicaciones y soluciones de código libre. Una alternativa a Amazon DynamoDB es Apache Cassandra, a Kinesis sería Apache Kafka, AWS Lambda tiene un funcionamiento similar a Open Whisk y Amazon S3 podría sustituirse por Hadoop. Amazon SNS y SQS permiten enviar notificaciones *push* y encolar mensajes, así que se podría desarrollar una solución que utilizara por ejemplo RabbitMQ y un servidor SMTP para envío de correos o utilizar APIs de aplicaciones como Slack para publicar mensajes.

Es interesante cómo Amazon añade una capa de seguridad e identidad[6]. En ella se gestionan los certificados X.509 y otras claves que pudieran utilizarse entre los dispositivos y la plataforma, además de definir las políticas de acceso para los usuarios (roles IAM y credenciales de usuario y cuentas de servicio). Este servicio es muy importante ya que el módulo de reglas envía los datos al resto de aplicaciones de AWS y necesita credenciales y permisos para ello, que se gestionan con esta capa de seguridad.

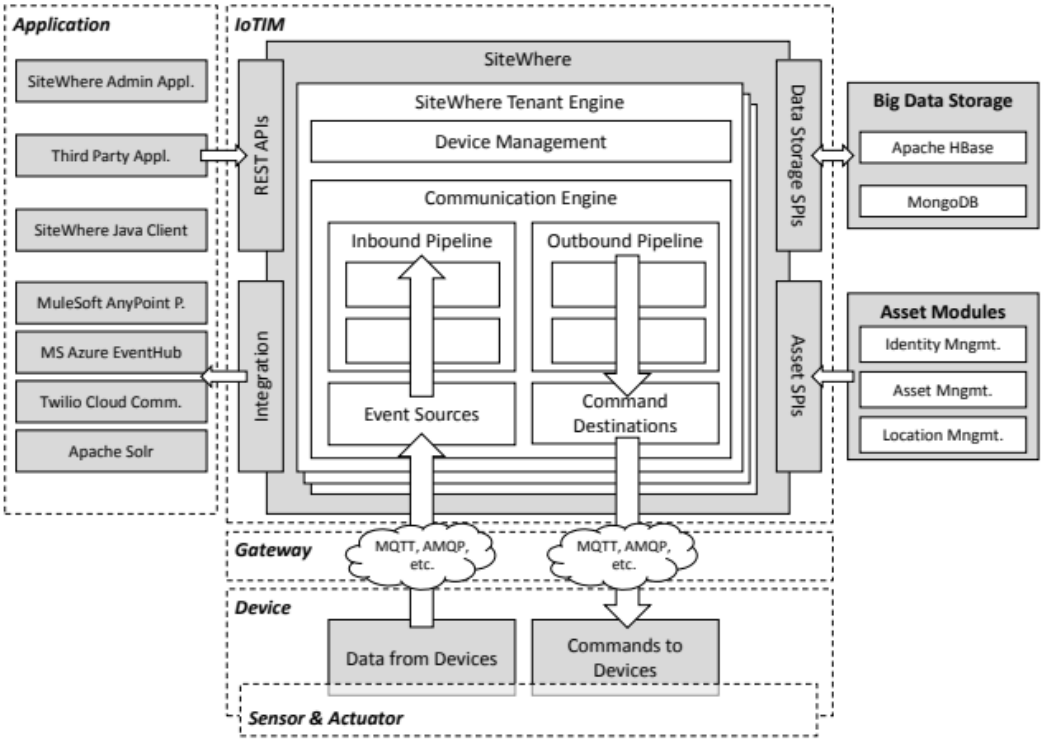


Figura 3.5: Arquitectura de SiteWhere IoT. Fuente: Guth et al. (2018).

Las alternativas libres que se contemplan en el artículo también tienen arquitecturas similares entre sí. En la Figura 3.5 se puede ver un esquema de SiteWhere. La idea es similar a la arquitectura propuesta para este trabajo y a la que las soluciones cloud utilizan. Los dispositivos utilizan *gateways* con distintos protocolos como MQTT o AMQP. Al igual que ocurría con AWS IoT, la comunicación entre la plataforma y los dispositivos es bidireccional, permitiendo la existencia de sensores y actuadores. Incluye un módulo para la gestión de dispositivos y distintas integraciones, como por ejemplo Apache Solr o EventHub de Azure. Incluye además varias APIs REST para integrar desarrollos de terceros o administrar la plataforma desde otras aplicaciones. El componente de almace-

[6] https://docs.aws.amazon.com/iot/latest/developerguide/iot-security.html

namiento lo resuelve con conectores para bases de datos como Apache HBase o MongoDB.

3.5.3. Ventajas del diseño propuesto

En la tabla 3.1 se pudo ver la gran diferencia en cuanto a las capacidades de las plataformas mencionadas y la propuesta en este trabajo. Sin embargo, no todos los desarrollos de un producto buscan utilizar herramientas muy completas, sino que en ocasiones es necesario partir de una pieza central sencilla pero que sea fácilmente modificable.

Sencillez vs completitud El principal objetivo que se tuvo en mente a la hora de diseñar la arquitectura era el hecho de mantenerla simple y que el esfuerzo de añadir componentes fuera el mínimo. Por ese motivo se utilizan herramientas bien conocidas (Emily Freeman, 2020) como son Kubernetes, Kafka y Go. Además, son herramientas relativamente nuevas con una gran comunidad que las mantiene actualizadas.

A una empresa que busque crear una prueba de concepto sencilla e ir construyendo *sprint* tras *sprint* le puede aportar más una base sólida y sencilla sobre la que partir. Deben ser capaces de entenderla rápidamente para poder comenzar a añadir funcionalidades e integrarla con el resto de sus desarrollos. Perderán un tiempo muy valioso si deben dedicar semanas al estudio de la plataforma o incluso cursar algún tipo de formación, como la que ofrecen la mayoría de productos en la nube. Además, podría ocurrir que tras esta inversión en el estudio de la herramienta pudieran darse cuenta de que no es realmente lo que necesitaban, ya sea por exceso de complejidad o falta de funcionalidades.

Soluciones *multi-cloud* Por otro lado, utilizar Kubernetes hace que la plataforma sea *multi-cloud*. Por lo tanto, no solo se va a poder migrar entre proveedores, sino que si se necesita una disponibilidad máxima se podría replicar entre varias cloud distintas. De esta forma se pueden evitar caídas del servicio si uno llegara a fallar o reducir costes gracias a las cuotas gratuitas que ofrecen. Además, permite que equipos que trabajan en un solo cloud puedan desplegarla en él directamente sin necesidad de duplicar servicios, como ocurriría si se utiliza AWS IoT pero el resto de los desarrollos se tienen en GCP.

Modificaciones e integraciones En cuanto a las integraciones y modificaciones, es cierto que otras opciones ofrecen multitud de conectores preconfigurados, pero siempre será más sencillo añadir una nueva integración en una base simple que modificar una plataforma tan grande como pudiera ser FIWARE o SiteWhere. Por ejemplo, añadir un campo nuevo al dato que llega a la plataforma o soportar actuadores sería cuestión de modificar una estructura o añadir un nuevo *endpoint* en el *router* del servidor. Para añadir microservicios bastaría con crear imágenes de Docker que expongan una API REST y añadir la llamada desde el recolector de datos u otros módulos. En la sección 5 se muestra cómo visualizar el dato entrante en tiempo real con apenas 50 líneas de Python.

Por otro lado, los componentes de FIWARE están escritos en varios lenguajes[7] que incluyen C++, Ruby, Java y Python, por lo que será necesario un equipo con conocimiento en varios lenguajes de programación para poder modificar el código. SiteWhere está escrito solamente en Java, pero compuesto por multitud de microservicios (en la Figura 3.6 se muestra un diagrama). Llegar a entender tal cantidad de elementos puede ser muy costoso y habrá muchas situaciones en las que no sea necesario tanta complejidad.

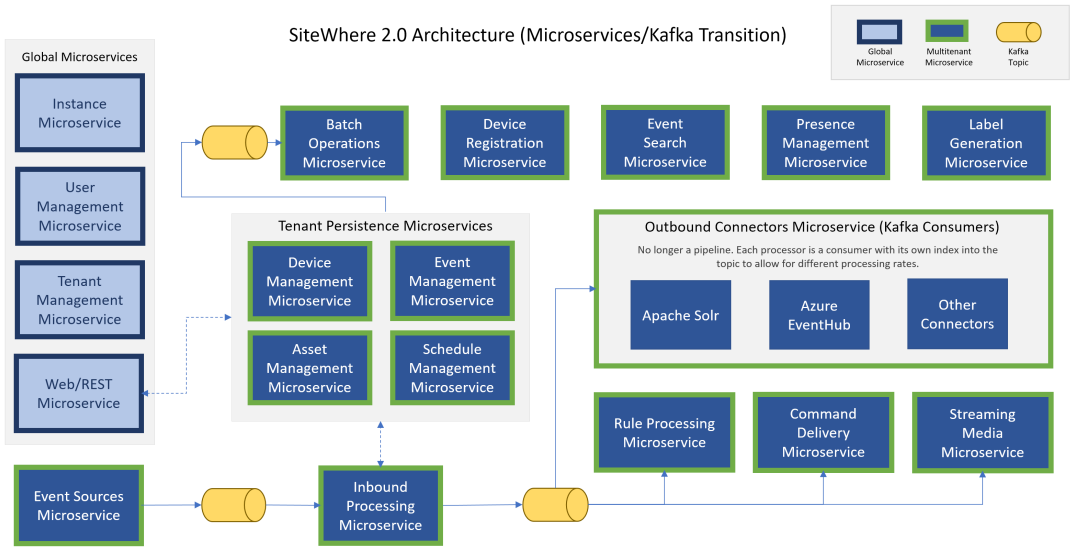


Figura 3.6: Diagrama de microservicios de la plataforma SiteWhere (Docs, 2021).

Casos de uso En cuanto a los casos de uso, FIWARE está orientado a empresas con casos de uso concretos que estén reflejados en la documentación de la herramienta: *smart cities*, *smart agrifood*, *smart robots*... Si se quisiera añadir un nuevo campo habría que

[7] https://github.com/Fiware

crear un *Smart Data Model*[8] y que el equipo de FIWARE lo apruebe mediante revisión manual.

En cuanto al mercado objetivo, FIWARE ha trabajado de forma conjunta con empresas como Red Hat o Atos e incluso ayudan a *startups* a adoptar la herramienta[9]. SiteWhere por el contrario está orientado a la comunidad, no a empresas. Las soluciones cloud sirven tanto para pruebas de concepto sencillas como productos completos para empresas y equipos. El diseño propuesto estaría orientado a empresas y grupos independientes que quieran crear un producto con total control y conocimiento sobre sus componentes.

3.6. Resumen del capítulo

En esta sección se ha establecido una arquitectura básica para una plataforma IoT sin restringirla a ningún tipo de herramienta concreta. Aún así se han mencionado una serie de proyectos de código libre que podrían utilizarse en cada uno de los componentes.

Se ha definido la arquitectura que se va a utilizar en la prueba de concepto, que contiene todos los componentes fundamentales e incluso un simulador de dispositivos. La elección de herramientas se ha hecho con el objetivo de que fueran de código libre y utilizando, en la medida de lo posible, proyectos de la CNCF[10] (*Cloud Native Computing Foundation*), ya que se caracterizan por ser escalables y poder desplegarse en nubes híbridas. Kubernetes y Prometheus son dos ejemplos.

Se ha podido comprobar lo sencillo que es incorporar nuevas herramientas de código libre a la arquitectura base gracias a los componentes que se han elegido y la multitud de aplicaciones que se podrían desarrollar a partir de ella.

Finalmente se ha incluido una comparación con plataformas IoT comerciales, comprobando que tanto entre ellas como con la arquitectura propuesta tienen muchos componentes en común. Se enumeran también las ventajas que tendría utilizar esta propuesta frente al resto de soluciones.

[8] https://www.fiware.org/developers/smart-data-models/
[9] https://www.fiware.org/community/fiware-accelerator-programme/
[10] https://www.cncf.io/

Capítulo 4

Implementación

4.1. Introducción

En esta sección se detalla el despliegue de los componentes de la plataforma y las herramientas utilizadas, enumerando las razones por las que se han escogido. Se utiliza Kubernetes como orquestador de contenedores. Para simplificar el desarrollo de las APIs REST, se ha optado por simular solamente sensores y no actuadores.

Para la implementación se ha seguido un esquema en microservicios y contenedores en lugar de una solución monolítica. Esto se debe a que si se pretende responder ante cargas del orden de las que se manejan en el campo del Big Data, debe ser posible escalar la plataforma de manera sencilla. Ya no solo eso, si se quiere añadir nuevas funcionalidades es más rápido y seguro hacerlo de esta forma[1].

Una opción sería desplegar cada servicio a mano en nodos individuales, entendiendo el nodo como una máquina virtual, contenedor o un servidor físico. Sin embargo, esta forma de proceder requeriría un mantenimiento a la larga y un tedioso despliegue. Habría que utilizar herramientas como Terraform y Ansible si se quisiera automatizar y replicar el despliegue. Una solución a este problema es utilizar un orquestador de contenedores, con el objetivo de automatizar tanto el despliegue como el mantenimiento y abstraer la plataforma del *hardware* sobre el que se instale. Los más conocidos son Docker Swarm[2] y

[1] https://www.redhat.com/es/topics/microservices

[2] https://docs.docker.com/engine/swarm/

Kubernetes[3] u Open Shift[4], la versión de Kubernetes propuesta por Red Hat.

Ambas opciones permiten escalar y manejar miles de contenedores y nodos de forma relativamente sencilla. Tanto Kubernetes como Swarm utilizan YAML como lenguaje para definir la configuración de los despliegues, lo cual permitirá aplicar control de versiones a la administración de los contenedores. Otro punto en común es que ofrecen opciones para la configurar cómo se exponen los servicios al exterior, balancear las cargas, definir un DNS para el clúster, despliegue continuo de aplicaciones o controlar la salud de los contenedores.

La principal diferencia se encuentra en que Kubernetes ofrece más herramientas y funcionalidades, además de tener una mayor comunidad. Una simple búsqueda en *Google Trends* muestra un interés de media de 56 para Kubernetes frente a 4 para Swarm en los últimos 5 años. Por otro lado, es un proyecto *graduado* de la CNCF y desarrollado en gran parte por Google, por lo que cuenta con un fuerte respaldo.

4.2. Introducción a Kubernetes

Para este despliegue se ha utilizado Kubernetes, un orquestador de contenedores para sistemas distribuidos. Sus principales características son:

- Manejar las máquinas donde se van a alojar los contenedores.
- Desplegar contenedores de forma sencilla.
- Ofrecer una alta disponibilidad.
- Poder escalar de forma rápida y automática.
- Ofrecer un *Disaster Recovery*, o recuperación de datos y servicios, tras un error de ejecución en los contenedores.

Estos cinco principios concuerdan perfectamente con lo que se busca de una plataforma de IoT que procese gran tráfico. La forma en que Kubernetes consigue estos objetivos es mediante cuatro componentes principales:

[3] https://kubernetes.io/

[4] https://www.redhat.com/es/technologies/cloud-computing/openshift

- **Nodo**: máquina física o virtual que contendrá el resto de componentes. La mayoría de los nodos serán de tipo *worker*, pero se utilizan nodos maestro para la administración del clúster.

- **Pod**: unidad mínima de Kubernetes. Almacena una aplicación ya sea en uno o más contenedores. Cada pod se ejecuta en un nodo. Una misma aplicación puede replicarse en varios pods para evitar caídas del servicio. Cada pod tiene una IP interna al clúster.

- **Servicio**: IP estática y puerto que se asocia a cad pod. En caso de que un contenedor del pod falle, se reinicia el pod y se despliega en el mismo o en un nodo distinto, cambiando su IP interna como consecuencia. Si otro pod estuviera utilizando esta IP dinámica en lugar de un servicio, la comunicación se perdería. Los servicios evitan este tipo de errores. Además, la IP del servicio no tiene por qué ser solamente interna al clúster. También se pueden utilizar para permitir el acceso desde el exterior.

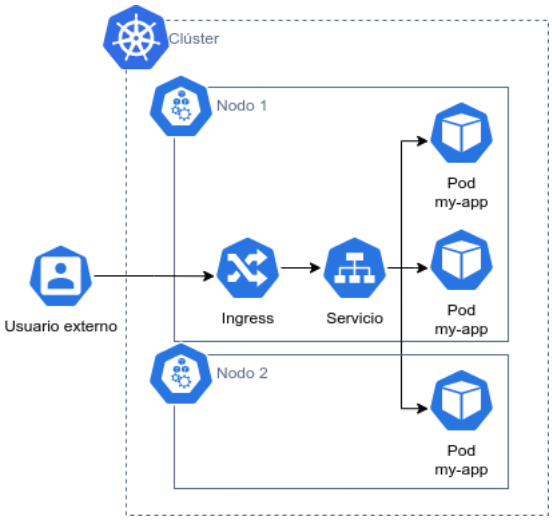
- **Ingress**: gestiona el acceso desde el exterior a los servicios del clúster, enrutando las peticiones a la par de ofrecer un balance de carga.

En la Figura 4.1a se puede ver una representación de cómo estos componentes se conectan entre sí.

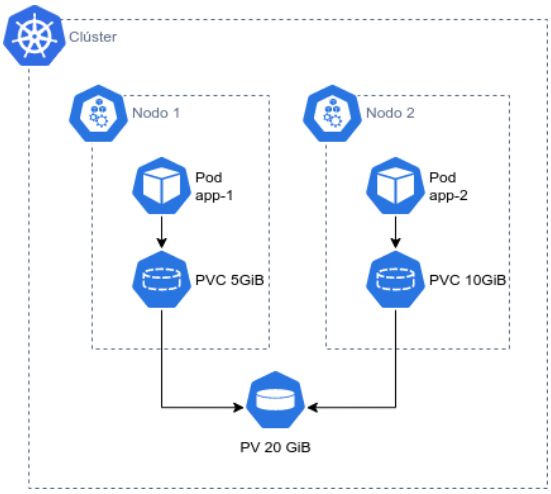
Otro componente fundamental si se quiere desplegar aplicaciones con estado, o *Stateful sets* para continuar con la nomenclatura de Kubernetes, son los **volúmenes persistentes** (PV). Al igual que ocurre con Docker, cada pod puede utilizar un volumen que actúa como sistema de ficheros compartido entre el pod y un nodo del clúster. De esta forma, si el pod se reinicia no pierde los datos que tenía guardados. Así se pueden desplegar bases de datos o sistemas de almacenamiento distribuido como HDFS, sobre Kubernetes. Cabe añadir que un PV puede ser también un almacenamiento cloud, como por ejemplo Amazon S3, no siempre tiene que ir ligado a un nodo.

Para que los desarrolladores de las aplicaciones no tengan que de definir a qué volumen se deben conectar, se han creado los *Persistent Volume Claims* (PVC), reclamaciones de PV. Así, si un pod está utilizando un volumen persistente alojado por ejemplo en Amazon S3 y el administrador del clúster de Kubernetes decide moverlo a Google Cloud Storage, la aplicación no se percata ni debe reiniciarse. Otra característica muy interesante de los

PVC es que dos pods pueden compartir un mismo PV siempre y cuando tengan espacio suficiente. Por ejemplo, en la Figura 4.1b el primer pod solicita un volumen de 5GiB mientras que el segundo necesita 10GiB. El administrador del clúster ha preparado un PV con 20GiB, así que ambos pueden utilizarlo pero sus archivos estarán en sistemas separados.



(a) Ingress permitiendo el acceso desde el exterior del clúster a un servicio desplegado sobre los pods que alojan la aplicación *my-app*.



(b) Arquitectura de dos aplicaciones *stateful set*, cada una en un pod distinto, que comparten un mismo volumen persistente mediante PVC.

Figura 4.1: Arquitectura básica de un clúster de Kubernetes.

Todos estos recursos se definen en archivos de configuración que utilizan tres campos principalmente:

- **Metadata**: un diccionario que identifica el recurso. Así, si otro recurso necesita comunicarse con él puede hacer una consulta (en un campo `selector`) sobre los valores de *metadata* de todos los recursos y seleccionar solamente aquellos que le interesen.
- **Spec**: definición de los contenidos del recurso. Qué imágenes utiliza en caso de ser un pod, qué otros recursos conecta, etcétera.
- **Status**: no se define sino que se genera automáticamente. Es el estado deseado del despliegue.

En cada clúster debe existir al menos un nodo **nodo maestro** que solamente contendrá los siguientes servicios:

- Un servidor API para conectarse desde un cliente, como `kubectl`[5], y administrar el clúster.
- Un *scheduler* que gestiona el despliegue de los pods en función de los recursos de cada nodo.
- Un *controller manager*, que detecta cuándo un pod muere y solicita al *scheduler* que lo disponibilice de nuevo.
- Una base de datos clave-valor llamada `etcd` que almacena todos los cambios en los estados del clúster.

Los nodos maestro necesitan menos recursos que los trabajadores ya que su única función es orquestar. Se pueden tener varios nodos maestro para evitar que el clúster se quede inoperativo.

Por último, Kubernetes tiene su propio gestor de paquetes llamado `Helm`, al igual que Debian tiene `apt` o Python `pip`. Los paquetes se denominan *charts* y consisten en plantillas con ficheros de configuración que despliegan varios componentes. Por ejemplo, se podría desplegar el stack de ELK[6] (ElasticSearch, Logstash y Kibana) para almacenar logs de todos los pods y visualizarlos, ejecutando símplemente:

```
1 helm repo add elastic https://helm.elastic.co
2 helm install --name elasticsearch elastic/elasticsearch
3 helm install --name kibana elastic/kibana
4 helm install --name logstash elastic/logstash
```

Si hubiera que hacerlo de forma manual habría que configurar cada despliegue por separado, crear servicios, volúmenes, reglas de replicación, etcétera.

Existen más componentes y herramientas, pero con los que se han introducido hasta ahora es suficiente para entender lo que se ha desplegado. Cabe añadir que para la prueba

[5] https://kubernetes.io/docs/reference/kubectl/kubectl/
[6] https://github.com/elastic/helm-charts

de concepto de la plataforma no se ha creado un clúster de varios nodos sino que se ha utilizado Minikube[7] para alojar un clúster de un nodo maestro y otro trabajador en local.

4.3. Preparación del entorno

En las siguientes secciones se indican los pasos a seguir para instalar cada componente por separado. Si se quisiera desplegar toda la arquitectura con un solo comando, se puede consultar directamente la sección 4.8.2. En cualquier caso, para poder desplegar estos componentes sobre un clúster local de Minikube, será necesario instalar las siguientes herramientas:

Minikube Para instalar Minikube basta con obtener el enlace de descarga desde su documentación:

https://minikube.sigs.k8s.io/docs/start/

Por ejemplo, en un sistema Linux con una arquitectura x86-64 basta con ejecutar en la terminal[8]:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/
   minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Es necesario tener instalado en la máquina alguno de los proveedores que Minikube utiliza para virtualizar: Docker, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox, o VMWare. En este caso se utiliza Docker[9]. El clúster se inicia con el comando

```
minikube start --driver=docker
```

En caso de que se disponga de suficiente CPU y memoria, se pueden ampliar los recursos de Minikube utilizando por ejemplo

```
minikube start --cpus=4 --memory=8g
```

[7] https://minikube.sigs.k8s.io/

[8] Estas instrucciones se han comprobado por última vez en junio de 2021. Se recomienda consultar la documentación de Minikube para confirmar que los enlaces sean los correctos.

[9] Las instrucciones de instalación de Docker se pueden encontrar en https://docs.docker.com/engine/install/.

Si la máquina no tiene suficientes recursos, el comando `minikube start` utiliza por defecto 2 CPUs y 2GB de memoria RAM.

En caso de que ocurra alguna clase de error en el clúster o se quiera comenzar desde el inicio, se pueden destruir todos los recursos con

```
minikube delete
```

A continuación sería necesario iniciar el clúster con `minikube start` de nuevo.

Kubectl Para administrar el clúster es necesario instalar `kubectl`. Se pueden seguir las instrucciones de instalación desde

https://kubernetes.io/es/docs/tasks/tools/install-kubectl/

Para comprobar que el clúster desplegado con Minikube funcione correctamente se puede ejecutar

```
kubectl get all
```

Y listar todos los componentes del clúster. Para instalaciones nuevas solamente debería aparecer el siguiente servicio:

```
NAME                 TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1     <none>        443/TCP   36s
```

En las siguientes secciones se utiliza un *namespace* distinto al por defecto. Para crearlo se debe ejecutar:

```
kubectl create namespace develop
```

Es **necesario** crear este *namespace* si se van a utilizar los comandos de las siguientes secciones.

Helm Al final de la sección se utiliza Helm para instalar todos los componentes con un solo comando. Las instrucciones de instalación se pueden encontrar en

https://helm.sh/docs/intro/install/

Para comprobar que se haya instalado Helm correctamente se puede ejecutar por ejemplo

```
helm search hub wordpress
```

Este comando listará todos los *charts* que contengan la palabra 'nginx' en su nombre. Si no se encontrara ninguno o se tuviera un error, la instalación será incorrecta.

4.4. Recolección de los datos

Para recoger las medidas de los sensores se ha creado una API REST. Esta interfaz acepta cualquier petición POST que contenga un campo `data` y otro `id`. La comunicación se realiza mediante HTTPS para asegurar la confidencialidad de ésta. Una vez recibe los datos, los publica en el clúster de Kafka con el `id` como clave del mensaje y `data` como el contenido.

4.4.1. Por qué utilizar una interfaz

La existencia de esta capa previa al *broker* de Kafka puede parecer innecesaria. Si un dispositivo tiene capacidad como para enviar peticiones POST a este *endpoint*, ¿por qué no publicar los mensajes en Kafka directamente?

El principal motivo es por facilitar el acceso de los dispositivos a la plataforma. El usuario que deba programar el envío de los datos no necesita saber cómo se distribuyen los mensajes dentro de un clúster de Kafka ni cómo utilizar sus interfaces. Otra razón es que no todos los frameworks y lenguajes de programación tienen conectores a Kafka actualizados a las últimas versiones del *broker*. Además, si en un cierto momento se quisiera cambiar el *broker* de mensajería, por ejemplo por uno *serverless* como podría ser PubSub, no haría falta modificar el software de los dispositivos.

Otra gran ventaja de utilizar una interfaz entre el cliente y el *broker* es que es mucho más sencillo añadir lógica en la ingesta sobre una API REST que sobre un software de terceros.

4.4.2. Explicación del código

El código correspondiente a esta API se ha subido a un repositorio en Gitlab[10]. Está programado en Go, siguiendo TDD (desarrollo basado en pruebas) y utiliza la integración continua de Gitlab para comprobar que no haya fallos en el código. Se ha intentado abstraer el código al máximo para que añadir o modificar un componente no afecte al resto. El inconveniente de hacerlo así es que se deben simular ciertas interfaces para ejecutar los tests, como por ejemplo la conexión con el clúster de Kafka.

Un ejemplo de esta situación es el fichero `collector_test.go` (Listing C.4), que comprueba que el servidor acepta los mensajes que cumplen la estructura requerida y, en caso contrario devuelve los mensajes y códigos de error adecuados. Todas estas pruebas son independientes de la conexión con Kafka, así que se simula la publicación con un objeto `StubPublisher` que implementa a `Publisher`, quien publicaría los mensajes en Kafka. Se entiende mejor sobre el código:

`Publisher` se define (en el Listing C.3) como una interfaz que debe tener los métodos `Publish` y `Destroy`. El primero será quien publique mensajes al *broker* (nótese que no depende de si es Kafka, PubSub u otro).

```
type Publisher interface {
  Publish(id string, payload interface{}) error
  Destroy() error
}
```

Listing 4.1: Publisher.

`CollectorServer` (en el Listing C.3) contiene un campo `Publisher` y tendrá un método `ServeHTTP` para poder utilizarlo con la librería `http` de Go, pero no es relevante para el ejemplo. Utilizando un publicador genérico se abstrae el tipo de broker sobre el que publicaría.

```
type CollectorServer struct {
  Pub Publisher
}
```

Listing 4.2: CollectorServer.

[10] https://gitlab.com/iok8s/collector

`StubPublisher` (en el Listing C.7) implementa a `Publisher` y tiene además un campo `Err` que va a permitir simular errores en la publicación de mensajes a Kafka.

```
type StubPublisher struct {
  Err error
}
func (s *StubPublisher) Publish(id string, payload interface{}) error {
  return s.Err
}
func (s *StubPublisher) Destroy() error {
  return nil
}
```

Listing 4.3: StubPublisher.

Con estas tres piezas ya es posible construir un test y simular por ejemplo que se devuelve el error adecuado cuando no se puede publicar en el *broker*:

```
func TestPublishError(t *testing.T) {
    jsonStr         := []byte(`{"data": "ok", "id": "3"}`)
  expectedStatus := http.StatusInternalServerError,
  expectedError  := errors.New("cannot publish message"),
  s := integration.StubPublisher{
    Err: expectedError,
  }
  server := &CollectorServer{&s}
  request, err := integration.MakeRequest("/collect", jsonStr)
  if err != nil {
    t.Errorf("got error while making the request, %v", err)
  }
  response := httptest.NewRecorder()

  server.ServeHTTP(response, request)

  assertStatus(t, response.Code, expectedStatus)
  assertErrorMessage(t, response.Body.String(), expectedError)
}
```

Listing 4.4: Ejemplo de cómo utilizar `StubPublisher` en un test. Las funciones `assertStatus` y `assertErrorMessage` se aseguran de que los dos parámetros que se le introducen sean iguales.

Obsérvese cómo en la línea 8 se construye el servidor con `StubPublisher`. Permite que en este test se pruebe el funcionamiento del servidor web y no del publicador. Quizá en otra aplicación se quisiera enviar todos los mensajes recibidos a un canal de Slack, por ejemplo. En ese caso solamente habría que cambiar el publicador de `CollectorServer`, pero no el resto del funcionamiento.

El paquete `testing` de Go y otros como `httptest` hacen que aplicar metodologías de Test-driven development (TDD), o 'desarrollo guiado por pruebas', sea muy sencillo y beneficioso. Para el publicador de Kafka[11] (en los Listing C.5 y C.6) se ha utilizado el paquete `segmentio/kafka-go` de entre las distintas opciones disponibles, ya que recibe mantenimiento y actualizaciones semanales y tiene una comunidad muy activa.

4.4.3. Simulación de sensores

Por otra parte, se ha desarrollado un pequeño módulo que simula mensajes de dispositivos (en el Listing C.1). Esto permite además calcular el tiempo promedio que lleva publicar un mensaje, gracias de nuevo al paquete `testing` y sus *Benchmarks*. Se ha definido un *benchmark* (en el Listing C.8) que ejecuta peticiones en bucle tantas veces como sea necesario para calcular un promedio con baja varianza, en este caso con `StubPublisher` en lugar de `KafkaPublisher`:

```
1  func BenchmarkMakeRequest(b *testing.B) {
2    d := emulator.NewGeneralDevice()
3    publisher := StubPublisher{
4      Err: nil,
5    }
6    server := &server.CollectorServer{Pub: &publisher}
7
8    for i := 0; i < b.N; i++ {
9      m := d.Measure()
10     p, err := d.GeneratePayload(m)
11     if err != nil {
12       b.Errorf("error generating payload, %v", err)
13     }
14     req, err := MakeRequest("/collect", p)
15     if err != nil {
```

[11] https://gitlab.com/iok8s/collector/-/blob/master/internal/server/kafka-publisher.go

```
16        b.Errorf("error generating the request %v", err)
17      }
18      response := httptest.NewRecorder()
19
20      server.ServeHTTP(response, req)
21      if response.Code != http.StatusAccepted {
22        b.Errorf("got status %v, want %v", response.Code, http.
      StatusAccepted)
23      }
24    }
25  }
```

Listing 4.5: BenchmarkMakeRequest. Se itera `b.N` veces para que el tiempo promedio tenga baja varianza.

4.4.4. Integración continua

Para asegurar que todos los *commits* que se hagan al repositorio hayan pasado los tests, se crea una *pipeline* en Gitlab CI/CD[12], definida en el Listing C.12. Ésta despliega un clúster de Kafka utilizando `docker-compose`, definido en el Listing C.13, con un nodo para Zookeeper, otro para Kafka y otro que ejecuta los tests sobre el repositorio. Todos los contenedores se conectan en una misma red para permitir la conexión entre ellos. La *pipeline* se da por finalizada cuando el comando `go test` del último contenedor finaliza. En caso de que el código de salida sea 0 la *pipeline* finalizará correctamente y en caso contrario será fallida y enviará un correo a quien haya ejecutado el *commit*.

Por último, se han creado dos ejecutables: uno para desplegar la API REST[13] y otro para simular dispositivos[14], ambos configurables. Se definen en los Listing C.9 y C.10 respectivamente.

Para la API se puede elegir en qué puerto se despliega, cuál es la dirección del clúster de Kafka, en qué topic publica, qué partición usa y cuál es el tiempo máximo de conexión con Kafka antes de cerrarla. Por ejemplo:

```
1 go run cmd/collector/main.go  -topic topic_test -partition 0 \
```

[12] https://gitlab.com/iok8s/collector/-/blob/master/.gitlab-ci.yml
[13] https://gitlab.com/iok8s/collector/-/blob/master/cmd/collector/main.go
[14] https://gitlab.com/iok8s/collector/-/blob/master/cmd/emulator/main.go

```
    -kafkaAddress localhost:9092 -deadline 5 -serverAddress :5000
```

Listing 4.6: Inicio de la API.

Para simular 4 dispositivos, durante 10 segundos y cada 20 milisegundos se usaría:

```
go run cmd/emulator/main.go -nDevices 4 -timeout 10 -maxTimeMsg 20
```

Listing 4.7: Simulación de dispositivos.

También se puede configurar la URL donde publica los mensajes, pero por defecto usa `https://localhost:5000/collect`.

4.4.5. Seguridad de la capa de transporte (TLS)

Para poder disponer de HTTPS en la API es necesario crear las claves y certificados necesarias con:

```
openssl req  -new  -newkey rsa:2048 \
   -nodes  -keyout cmd/collector/localhost.key \
   -subj /CN=localhost/ \
   -out cmd/collector/localhost.csr

openssl  x509  -req  -days 365  -in cmd/collector/localhost.csr  -
   signkey cmd/collector/localhost.key  -out cmd/collector/localhost.crt
```

Listing 4.8: Creación del certificado para TLS. En este ejemplo se hace para `localhost`.

Como el certificado es auto-firmado, ha sido necesario ignorar la verificación en el código del emulador (Listing C.10):

```
http.DefaultTransport.(*http.Transport).TLSClientConfig = &tls.Config{
   InsecureSkipVerify: true}
```

Listing 4.9: Cambio de la configuración del emulador para ignorar la verificación de certificados.

4.4.6. Métricas para Prometheus

Para comprobar que la aplicación funcione de forma correcta y visualizar su estado se va a utilizar Prometheus. En la sección 4.7.1 se detalla su funcionamiento, pero de forma

resumida, es una herramienta que hace *scrapping* a varios objetivos que exponen métricas en un *endpoint* `/metrics`. Para conseguir estas métricas se usa la librería de Prometheus para Go[15].

Se crean tres métricas: el número total de operaciones finalizadas, el número actual de peticiones que se están procesando y un histograma con el tiempo que se tarda en procesar cada petición. Se definen de la siguiente forma:

```
import (
  "github.com/prometheus/client_golang/prometheus"
  "github.com/prometheus/client_golang/prometheus/promauto"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
  opsProcessed = promauto.NewCounter(prometheus.CounterOpts{
    Name: "collector_processed_ops_total",
    Help: "The total number of processed messages",
  })
  opsCurrent = promauto.NewGauge(prometheus.GaugeOpts{
    Name: "collector_current_ops",
    Help: "The current number of requests",
  })
  processingTime = prometheus.NewHistogram(prometheus.HistogramOpts{
    Name:    "collector_processing_time_seconds",
    Help:    "The processing time of messages in seconds.",
    Buckets: prometheus.LinearBuckets(0.005, 0.005, 10),
  })
)
```

Y se pueblan desde `CollectorServer` de la siguiente forma:

```
func (c *CollectorServer) dataHandler(w http.ResponseWriter, r *http.
    Request) {
  timer := prometheus.NewTimer(processingTime)
  defer timer.ObserveDuration()

  opsProcessed.Inc()
  opsCurrent.Inc()
```

[15] https://github.com/prometheus/client_golang

```
7   defer opsCurrent.Dec()
8
9   // Resto de procesado de la peticion
10 }
```

Lo que se hace es:

1. Iniciar un *timer* que inicia una cuenta progresiva.
2. Indicarle al *timer* con un `defer` que cuando la función termine envíe el resultado a Prometheus con la función `ObserveDuration`.
3. Incrementar el número de operaciones totales y de operaciones concurrentes.
4. Utilizar de nuevo un `defer` para cuando finalice la petición decrementar las operaciones concurrentes.

Por último, se expone el *endpoint* de `/metrics` junto con el que procesa las peticiones de recolección de mensajes:

```
1 func (c *CollectorServer) ServeHTTP(w http.ResponseWriter, r *http.
     Request) {
2   router := http.NewServeMux()
3   router.Handle("/collect", http.HandlerFunc(c.dataHandler))
4   router.Handle("/metrics", promhttp.Handler())
5
6   router.ServeHTTP(w, r)
7 }
```

Se puede ver este apartado en el Listing C.3.

4.4.7. Despliegue en Kubernetes

Este despliegue se debe hacer a continuación de disponibilizar el clúster de Kafka, que se detalla en la sección 4.6. En caso contrario el contenedor se reiniciará hasta que consiga conectarse con un clúster de Kafka.

La configuración para el despliegue se muestra en el Listing B.1. En él se define el nombre del despliegue una `label` *collector* que será la que utilice el servicio para encontrar

los pods a los que debe redirigir las peticiones. En el campo `spec.template.spec` se indica la imagen de Docker que se va a utilizar. En este caso es una imagen con el código de Gitlab que se introdujo en esta sección y que se puede consultar en el Listing C.11. Se expone el puerto 5000, ya que toma los parámetros por defecto que son:

- API expuesta en el puerto 5000.
- Todos los mensajes se escriben en la partición 0, que es la única con la que se va a trabajar.
- Tras 5 segundos sin recibir un mensaje cierra la conexión con Kafka.
- El nombre del topic donde publica es `topic_test`.
- La conexión con el clúster de Kafka se busca en la dirección `localhost:9092`.

Este último parámetro se sobrescribe en la última línea del Listing. Se introduce la dirección del clúster de Kafka que se despliega en la próxima sección junto con el puerto.

Para poder publicar enviar mensajes a la API es necesario tener acceso desde fuera del clúster. Para ello se define el servicio del Listing B.2. El campo `spec.selector` es quien decide sobre qué pods actuar. Se introduce la label `app` cuyo valor es `collector`, que se definió en el deployment de la API. Mapea el puerto 5000 del pod al 5000 del servicio y hace una redirección al puerto 30500 del nodo. A este puerto se le asigna el nombre de `https`, que más adelante se utilizará.

Es necesario advertir en este punto que esta no es una configuración aconsejable, ya que exponer el puerto del nodo puede suponer que si el nodo se reinicia y cambia de IP se pierda el acceso, por no hablar de que es un riesgo para la seguridad del clúster y no balancea la carga. Además, el componente *ingress* se ha diseñado para evitar este tipo de prácticas. Sin embargo se ha hecho así puesto que esto es una prueba de concepto y además se trabaja sobre Minikube, donde la configuración del *ingress* puede dar bastantes problemas. Es también la forma más común de proceder cuando se desarrolla una aplicación, pero no cuando se publica en entornos de producción.

Finalmente, para poder recoger las métricas del pod se crea un *service monitor*. Prometheus tiene una funcionalidad llamada *service discovery* que se basa en buscar todos los

service monitor del clúster que contengan una *label* con clave `release` y valor `prometheus`. Este *service monitor* se define en el Listing B.3, que principalmente recoge tres configuraciones:

- El servicio que monitoriza se indica en el campo `spec.selector.matchLabels.app`.
- Dentro de `endpoint` se especifica:
  - El puerto que contiene el *endpoint* `/metrics`.
  - Cada cuánto tiempo se recogen las métricas.
  - La configuración TLS, que en este caso solo se especifica que no verifique el certificado (ya que es autofirmado).
  - El tipo de comunicación que va a utilizar, que se especifica HTTPS ya que por defecto usa HTTP.

Para desplegar el pod simplemente se ejecuta:

```
kubectl apply -f collector.yml -n develop
```

Se ha añadido el parámetro `-n` para utilizar un *namespace* llamado *develop*. Los *namespaces* no son más que grupos de componentes que permiten aislarlos para distinguir entre entornos de desarrollo y producción o distintos equipos.

El fichero `collector.yml` contiene los tres Listings que se han mencionado en la sección. En el repositorio de Gitlab del trabajo[16] se especifican también los comandos necesarios para publicar mensajes y consumirlos mediante un consumidor de Kafka para terminal.

4.5. Módulo de reglas

El módulo de reglas desarrollado consiste dos partes fundamentales: una API REST que gestiona la creación, modificación y almacenamiento de reglas y una interfaz web que permite su creación y visualización de forma sencilla.

[16] https://gitlab.com/iok8s/kubernetes-cluster/-/blob/master/collector.md

Al igual que con la API de recolección de datos, en esta parte se utiliza Go para crear el *backend*. Se utilizan además las plantillas, o acciones, de Go[17] para generar la interfaz web. Esta es una funcionalidad muy utilizada y que se verá también a la hora de crear un *chart* de Helm en la sección 4.8.2.

Este módulo está diseñado para recibir el identificador del dispositivo y el contenido del campo `data` que la API de recolección extrae de cada mensaje.

Como el resto de desarrollos del trabajo, el repositorio correspondiente al módulo de reglas se puede encontrar en `https://gitlab.com/iok8s/rule-engine`.

4.5.1. Motor de reglas

La pieza básica de este módulo consiste en un motor de reglas, encargado de modificar la base de datos correspondiente. En este caso concreto se ha utilizado Redis[18], una base de datos en memoria utilizada mayormente como caché o como *broker* de mensajería. Se ha elegido esta por su gran velocidad de lectura y escritura (Tang and Fan, 2016), bajo consumo de recursos y su capacidad para escalar tanto de forma horizontal como vertical.

En el Listing D.1 se define la interfaz `RuleEngine` y la estructura `Rule`, que deben seguir todas las reglas. `RuleEngine` contiene un único atributo y es la conexión con la base de datos. Se utiliza de nuevo TDD para desarrollar el código. Las pruebas se encuentran en el Listing D.2. Para simular la conexión con la base de datos se ha utilizado `miniredis`[19].

Las reglas se componen de tres campos:

1. `Blocked`: valor booleano que indica si el dispositivo está bloqueado o no. Será de utilidad si se quiere combinar este módulo con otras herramientas existentes como podría ser LDAP (Protocolo Ligero de Acceso a Directorios) o Fail2Ban[20]. En Lopez-Araiza and Cankaya (2017) se puede encontrar una introducción y comparación de Fail2Ban con otras herramientas de análisis forense.

2. `Fields`: lista de campos obligatorios que deben contener los mensajes.

[17] `https://golang.org/pkg/text/template/`
[18] `https://redis.io/`
[19] `https://github.com/alicebob/miniredis/`
[20] `https://www.fail2ban.org`

3. `MaxSize`: tamaño máximo de la petición. De gran utilidad para prevenir sobrecargas en la API.

Cada regla se asocia además con una ID de tipo cadena de caracteres. Las funciones de este motor son:

- Guardar una regla: `setRule(id string, rule Rule) error`.
- Consultar una regla: `getRule(id string) (Rule, error)`.
- Listar todas las reglas: `listRules() (map[string]Rule, error)`.
- Comprobar si un ID está bloqueado: `idIsBlocked(id string) (bool, error)`.
- Comprobar si un mensaje contiene los campos necesarios: `blockedByFields(id string, fields []string) (bool, error)`.
- Consultar el campo `maxSize` para un ID dado: `getMaxSize(id string) (int, error)`.

En este listado se ha seguido el formato de definición de funciones en Go: nombre de la función, entradas y salidas.

4.5.2. Servidor

API REST

En el Listing D.3 se define la interfaz para la API con los siguientes *endpoints*:

- `GET /auth/IDENTIFICADOR_DISPOSITIVO`: recibe el identificador del dispositivo y un mensaje y comprueba si está bloqueado o no según las reglas guardadas. En caso de que la regla no exista, se supone que el dispositivo no está bloqueado.
- `GET /rule/IDENTIFICADOR_DISPOSITIVO`: devuelve la regla correspondiente al identificador del dispositivo.
- `POST /rule/IDENTIFICADOR_DISPOSITIVO`: guarda la regla para el dispositivo. Se espera un contenido en formato JSON con los campos `blocked`, `fields` y `maxSize`.

- `GET /rules`: devuelve una lista con todas las reglas de la base de datos.

Todos estos *endpoints* se prueban en el Listing D.4. En el Listing D.5 se tiene el código con el que se genera el ejecutable, que despliega el servidor descrito en esta sección junto con un motor de reglas configurable mediante variables de entorno. Las variables que se utilizan son la dirección del clúster Redis, el puerto y la contraseña para la conexión.

Web estática

Para las peticiones recibidas en la raíz de la dirección URL, se genera un sitio web estático utilizando las plantillas de Go. A grandes rasgos, lo que permiten estas plantillas es generar cadenas de texto en función de parámetros que se introduzcan desde Go. En este caso las cadenas que se generan son filas de una tabla donde se listan las reglas. La plantilla se carga con:

```
var tpl = template.Must(template.ParseFiles("assets/index.html"))
```

Y se rellena con:

```
results := Results{
    Total: len(rules),  // Numero total de reglas
    Rules: rules,       // Lista de reglas
  }
tpl.Execute(w, results)
```

En este caso `w` es el `ResponseWriter` de la petición[21]. En el código HTML se define la lógica de estas plantillas entre corchetes de la siguiente manera:

```
<p>En total hay {{ .Total }} reglas</p>
```

para acceder al campo `Total` que se definió en la variable `results` y utilizar su valor para completar un párrafo. Las plantillas no solo se utilizan para enviar datos desde el *backend* al *frontend*, sino que se puede generar el contenido con, por ejemplo, bucles:

```
<tbody>
    {{ range $key, $rule := .Rules }}
    <tr>
        <th scope="row">{{ $key }}</th>
```

[21] https://golang.org/pkg/net/http/#ResponseWriter

```
        <td>{{ $rule.Blocked }}</td>
        <td>
            {{ range $field := $rule.Fields }}
            <p>{{ $field }}</p>
            {{end}}
        </td>
        <td>{{ $rule.MaxSize }}</td>
    </tr>
    {{end}}
</tbody>
```

En este ejemplo del Listing D.6 se utiliza la función `range` para generar tantas filas como reglas se tengan en la base de datos y tantos elementos por celda como campos existan en esa regla.

La interfaz web utiliza los *endpoints* de la API REST para interactuar con la base de datos. Por ejemplo, el formulario de creación de reglas hace una petición POST a `/rule/`. Por defecto los formularios se envían codificados en `x-www-form-urlencoded`. Por ello, se ha modificado el funcionamiento del formulario para que se envíe un JSON en su lugar. El código Javascript se encuentra en el Listing D.7.

4.5.3. Uso de la aplicación

Creación de reglas

Para crear una regla utilizando la API REST se puede hacer una petición POST al *endpoint* `/rule/` con el ID del dispositivo y la definición de la regla en el cuerpo del mensaje:

```
curl -d '{"blocked": false, "fields": ["temperature", "humidity"], "
    maxSize": 205}' -H "Content-Type: application/json" -X POST http://
    localhost:3000/rule/my-id
```

Por otro lado, también se podría utilizar la interfaz web y crear la regla desde un formulario, como se muestra en la Figura 4.2.

Para modificar una regla basta con crear una nueva con los nuevos parámetros y el mismo identificador de dispositivo.

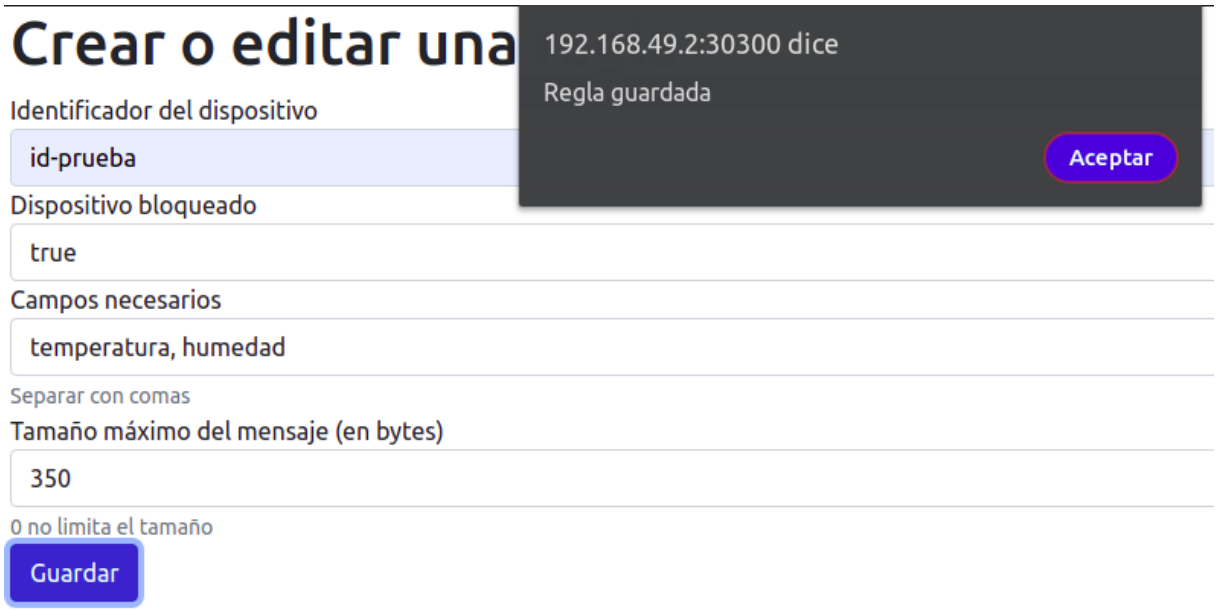


Figura 4.2: Creación de una regla desde la interfaz web.

Consultar las reglas almacenadas

Para listar las reglas utilizando la API REST se consulta mediante GET el *endpoint* `/rules`:

```
curl http://localhost:3000/rules
```

También es posible consultar la regla para un dispositivo en concreto especificando su identificador en la URL de la petición:

```
curl http://localhost:3000/rule/my-id
```

En la interfaz web se listan todas las reglas en la página principal, como se muestra en la Figura 4.3.

Consulta de la autorización

Para comprobar si un dispositivo tiene permitido publicar en la plataforma y que además su mensaje cumple con los requisitos establecidos en la regla, se debe consultar el *endpoint* `auth` con una petición GET y el identificador:

```
curl -d '{"temperature": 23.5, "humidity": 80}' -H "Content-Type:
    application/json" -X POST http://localhost:3000/auth/my-id
```

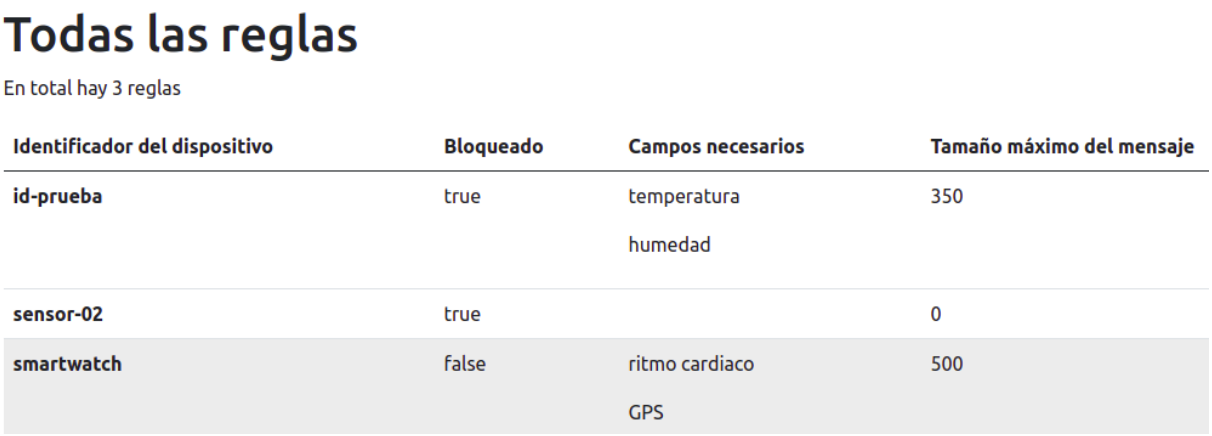


Figura 4.3: Listado de reglas desde la interfaz web.

El código de estado de la respuesta será 403 (*forbidden*) en caso de que no cumpla con los requisitos.

4.5.4. Despliegue en Kubernetes

La imagen de Docker se crea a partir del Listing D.9. Para crear un pod en Kubernetes se procede de igual forma que con la API de recolección de datos. Por lo tanto, para desplegar los pods del módulo de reglas bastará con ejecutar:

```
kubectl  apply -f rule_engine/rest-api.yaml -n develop
```

Este fichero de configuración se puede encontrar en el Listing B.7.

Despliegue del clúster de Redis

Para desplegar el clúster de Redis se ha optado por una instalación manual. Existen algunos operadores[22] pero aún no tienen versiones estables. Por esta razón y por tener un mayor control y capacidad de personalización, se ha optado por definir desde cero.

El clúster tendrá un número N de nodos trabajadores, uno de ellos *master*, y M nodos Sentinel, definidos con el número de réplicas de los pods. Los primeros negociarán entre ellos para determinar quién toma el rol de *master* en caso de que éste se destruya, mientras que los Sentinel monitorizarán el clúster para asegurar que la elección se realice sin problemas. También se puede utilizar Sentinel para informar a los administradores del

[22] https://operatorhub.io/operator/redis-operator

sistema en caso de que hubiera algún tipo de error. Se recomienda utilizar al menos tres nodos por clúster[23], aunque para una prueba de concepto con menos serviría. Los ficheros de configuración están preparados para admitir un número flexible de réplicas.

La configuración común a los nodos y a Sentinel se define en un *configmap*. No se adjunta en los apéndices por su gran extensión (unas 2000 líneas), pero sí que se pueden encontrar las líneas más relevantes o consultar directamente sobre el repositorio[24]. Los nodos se despliegan con el código del Listing B.8 y los de Sentinel en el B.9. Se han utilizado *StatefulSets* y volúmenes persistentes porque Redis es una aplicación con estado y necesita persistencia del almacenamiento, no como las APIs que se han desarrollado.

Han sido realmente útiles las plantillas de Helm para adecuar la configuración de Sentinel al número de réplicas que se utilicen en el despliegue. Por otro lado, se ha tenido que añadir un bucle en el inicio de los nodos Sentinel que espere a que los nodos trabajadores reciban peticiones ya que en caso contrario se quedaría en un estado *CrashLoopBackoff*.

4.6. Despliegue del clúster de Kafka

En lugar de definir el clúster de Kafka desde cero se ha utilizado un operador de Kubernetes. Estos operadores no son más que recursos personalizados para manejar las aplicaciones y sus componentes[25] y se suelen instalar con Helm. Una explicación más detallada del despliegue que se va a realizar se puede encontrar en el repositorio del trabajo[26].

Para instalar el operador se ejecuta:

```
1 kubectl create -f 'https://strimzi.io/install/latest?namespace=develop'
    -n develop
```

Este operador va a permitir que en archivos de configuración, como el del Listing B.4, se puedan utilizar recursos como el 'Kafka', que se utiliza en las líneas 2 y 3 pero que

[23] https://docs.redislabs.com/latest/rs/administering/designing-production/hardware-requirements/

[24] https://gitlab.com/iok8s/charts/-/blob/master/iok8s/templates/rule_engine/redis-configmap.yaml

[25] https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

[26] https://gitlab.com/iok8s/kubernetes-cluster/-/blob/master/strimzi.md

en la versión base de Kubernetes no existen. Se despliega entonces un clúster de un solo broker con este archivo de configuración ejecutando:

```
kubectl apply -f kafka-persistent-single.yaml -n develop
```

Tras ello estarán disponibles los cuatro pods que se han definido junto con el operador:

```
 ~/gitlab/kubernetes > kubectl get pods -n develop
NAME                                          READY   STATUS    RESTARTS
     AGE
cluster-01-entity-operator-9cd7567d9-gtslr    3/3     Running   2
     4m42s
cluster-01-kafka-0                            1/1     Running   0
     5m9s
cluster-01-kafka-exporter-7dd6fc8f46-g7ggb    1/1     Running   1
     5m43s
cluster-01-zookeeper-0                        1/1     Running   0
     6m32s
strimzi-cluster-operator-957688b5c-45w98      1/1     Running   0
     16m
```

El primero de ellos es el Operador Entidad, encargado de crear los topics y usuarios del clúster. El segundo es el broker de Kafka, cuyas métricas se exportan de formato JMX (el predeterminado de Kafka) a uno que Prometheus pueda consultar mediante el pod *kafka-exporter*. Finalmente se tiene un único pod con Zookeeper. Se ha configurado el nombre del clúster como 'cluster-01', por eso cada pod comienza con ese prefijo.

Al igual que se hizo con la API, es necesario crear un servicio y un *servicemonitor* sobre el *kafka-exporter* para exponer las métricas y que Prometheus pueda consumirlas. La configuración de estos servicios se muestra en el Listing B.5 y B.6 respectivamente. Se crean mediante:

```
kubectl apply -f kafka-exporter-service.yaml -n develop
```

El fichero `kafka-exporter-service.yaml` contiene los dos Listing que se acaban de mencionar.

Para este ejemplo solamente se ha usado un broker y un nodo de Zookeeper, pero Strimzi permite desplegar tantos nodos como sea necesario y se puede usar el autoescalado nativo de Kubernetes para gestionar el clúster en función del tráfico.

Para probar que el clúster de Kafka esté correctamente configurado se puede iniciar un productor y un consumidor y enviar mensajes desde la terminal. Se usa `kubectl run` para desplegar dos pods con la imagen de Strimzi con la versión 2.7.0 de Kafka[27], con los nombres `kafka-producer` y `kafka-consumer` respectivamente. En ellos se ejecutan los scripts que vienen por defecto en la instalación de Kafka `kafka-console-producer.sh` y `kafka-console-consumer.sh`:

```
1 kubectl -n develop run kafka-producer -ti --image=quay.io/strimzi/kafka
    :0.22.1-kafka-2.7.0 --rm=true --restart=Never -- bin/kafka-console-
    producer.sh --broker-list cluster-01-kafka-bootstrap:9092 --topic my-
    topic
2
3 kubectl -n develop run kafka-consumer -ti --image=quay.io/strimzi/kafka
    :0.22.1-kafka-2.7.0 --rm=true --restart=Never -- bin/kafka-console-
    consumer.sh --bootstrap-server cluster-01-kafka-bootstrap:9092 --
    topic my-topic --from-beginning
```

Si la configuración es correcta, deberían leerse desde el consumidor los mensajes del publicador con no muchos milisegundos de latencia, dependiendo de los recursos asociados al clúster.

4.7. Recogida y visualización de métricas

Para visualizar el estado del clúster de Kubernetes y de las aplicaciones que contiene se han utilizado dos herramientas: Prometheus[28] y Grafana[29]. La primera permite recoger métricas y alertar en caso de que algún servicio falle o comience a escasear algún recurso del clúster. La segunda sirve para crear *dashboards* personalizados con multitud de fuentes de datos, aunque en este caso se usa Prometheus únicamente.

4.7.1. Prometheus

La idea tras Prometheus es muy sencilla: recoger dato de una serie de *endpoints* (*targets*) y almacenarlo. Este dato ha deberá cumplir los formatos de las métricas de Pro-

[27] quay.io/strimzi/kafka:0.22.1-kafka-2.7.0
[28] https://prometheus.io/
[29] https://grafana.com/

metheus[30], que son contadores, *gauge*, histogramas y resúmenes.

Para recoger las métricas, Prometheus usa un componente que mediante peticiones PULL consulta cada *endpoint* y las almacena en una base de datos como series temporales. A su vez, dispone de un componente que expone un servidor HTTP que recibe una consulta en lenguaje PromQL (lenguaje de consultas personalizado para Prometheus), la ejecuta sobre la base de datos y devuelve su resultado. Otras herramientas, como es el caso de Grafana, disponen de conectores preconfigurados que utilizan este servidor HTTP para consumir las métricas. En la Figura 4.4 se puede ver un esquema de los componentes que forman la herramienta.

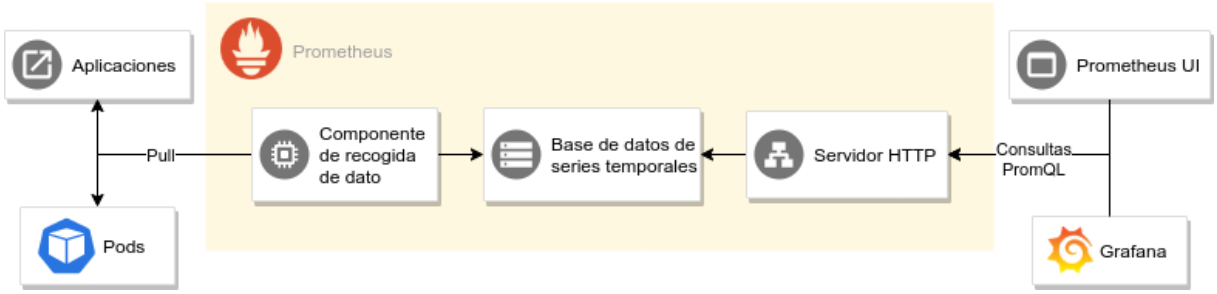


Figura 4.4: Esquema del funcionamiento de Prometheus.

Desplegar Prometheus en Kubernetes es muy sencillo. Al igual que con Kafka, se utiliza un operador que se instala con los siguientes comandos:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo add stable https://charts.helm.sh/stable
helm repo update
helm install prometheus prometheus-community/kube-prometheus-stack -n develop
```

No solo se crea un pod con el operador (`prometheus-kube-prometheus-operator`), sino que se crea:

- `alertmanager-prometheus-kube-prometheus-alertmanager-0`: para lanzar y gestionar alertas.
- `prometheus-grafana`: *dashboard* de Grafana.

[30] https://prometheus.io/docs/concepts/metric_types/

- `prometheus-kube-state-metrics`: para publicar el estado del clúster con el formato de métricas de Prometheus.
- `prometheus-prometheus-kube-prometheus-prometheus-0`: sería el servidor de Prometheus como tal, con el componente de recogida de datos, la base de datos y el servidor HTTP.
- `prometheus-prometheus-node-exporter-nswdn`: para exportar métricas del propio hardware del nodo donde se ha desplegado, siempre y cuando use un kernel de Unix.

Tan pronto como estos recursos estén disponibles, recogerá métricas del nodo y de los los *servicemonitor* que contengan la *label release* como `prometheus`, tal y como se configuró en las secciones del recolector de datos y de Kafka.

Para ver que se recogen las métricas correctamente se puede acceder a la UI de Prometheus y lanzar una consulta sencilla. Para ello se debe crear un servicio que exponga un puerto en el nodo. No es necesario crear un fichero de configuración, sino que simplemente se ejecuta:

```
~/gitlab/kubernetes > kubectl expose service prometheus-kube-prometheus-prometheus --type=NodePort --target-port=9090 --name=prometheus-server-np
service/prometheus-server-np exposed
```

Accediendo ahora a la IP de Minikube junto con el puerto se llega a la interfaz de Prometheus. Otra forma más sencilla de acceder es ejecutando:

```
minikube service prometheus-server-np -n develop
```

Esta opción incluso abre el navegador automáticamente. Una vez en la interfaz se comienza a escribir el campo que se desee y se autocompleta con las opciones disponibles. Por ejemplo, se puede ver el uso de CPU por cada uno de los contenedores, como en la Figura 4.5.

4.7.2. Grafana

Aunque es posible visualizar las métricas desde Prometheus UI, Grafana permite crear *dashboards* con multitud de paneles, dato en tiempo real y fuentes distintas. Además, estos

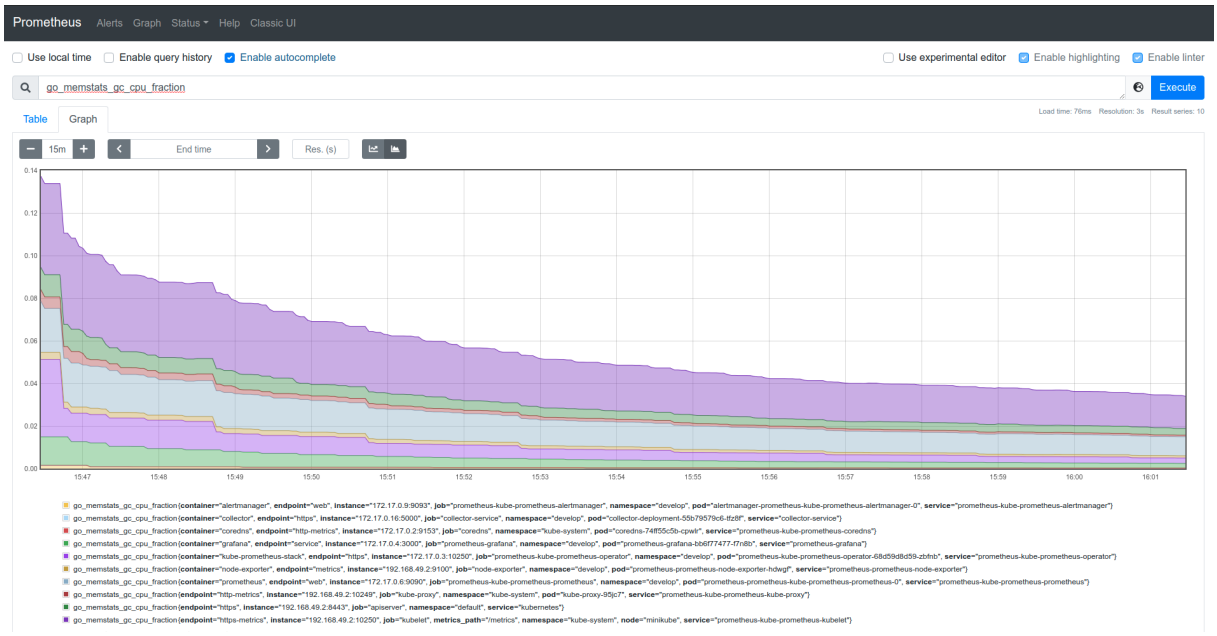


Figura 4.5: Consulta en Prometheus UI sobre el consumo del CPU por cada uno de los contenedores.

paneles se pueden personalizar sobre la interfaz directamente, arrastrando con el ratón o seleccionando el formato que mejor se ajuste a las necesidades de cada situación, sin necesidad de escribir código.

La comunidad de Grafana publica *dashboards* para las fuentes de datos más comunes[31]. Aunque en la instalación del operador de Prometheus ya se incluyen algunos *dashboards* para visualizar las métricas del clúster, para este trabajo me he basado en uno distinto aunque con paneles similares[32]. Le he añadido en la parte superior las métricas de la API además de las del clúster de Kafka. En la Figura 4.6 se puede ver la pantalla principal[33]. Para acceder a ella en caso de que se pida usuario y contraseña, deberá introducirse 'admin' y 'prom-operator' respectivamente.

En la parte inferior de la imagen aparecen más filas de paneles (*Deployments*, *Node*, *Pods* y *Containers*). Éstas se pueden ver expandidas en la Figura 4.7.

La configuración de los paneles es muy sencilla. En la Figura 4.8 se puede ver cómo se ha creado el de operaciones totales en la API. En el centro se tiene una previsualización del

[31] https://grafana.com/grafana/dashboards/

[32] https://grafana.com/grafana/dashboards/6417

[33] En https://gitlab.com/iok8s/kubernetes-cluster/-/blob/master/dashboard.json se puede descargar el fichero JSON del *dashboard* e importarlo en Grafana.

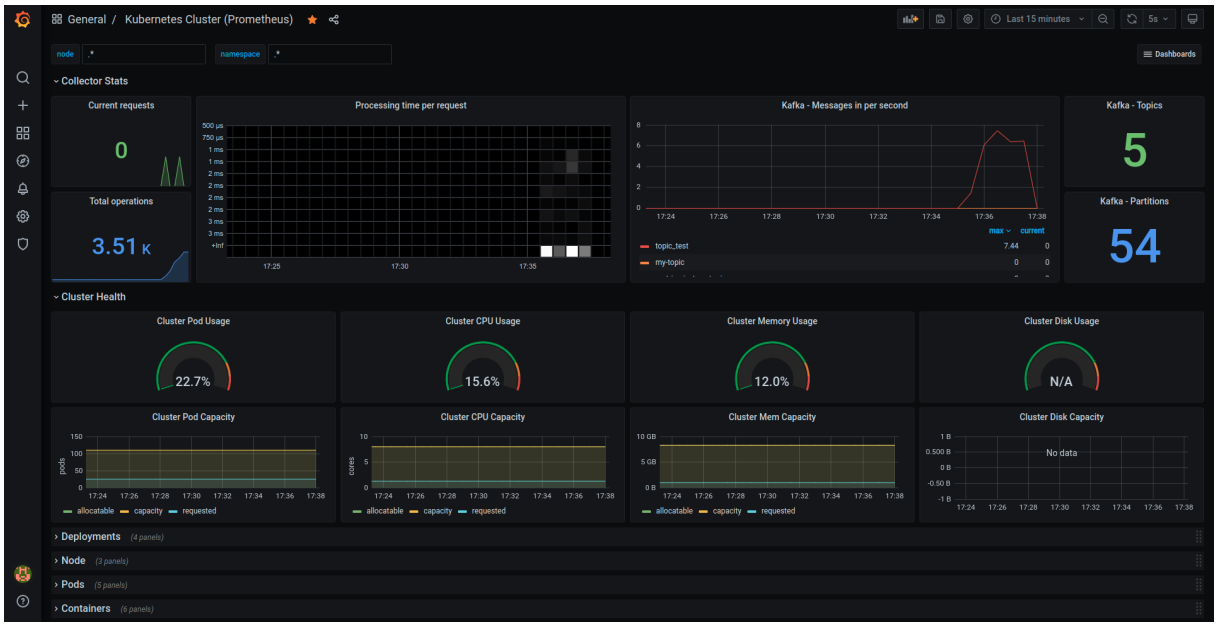


Figura 4.6: *Dashboard* con las métricas de la API de recolección de datos, el clúster de Kafka y el estado del clúster de Kubernetes.

panel con los ajustes seleccionados. En la parte inferior se escribe la consulta en formato PromQL. Sobre la consulta se selecciona la fuente de datos, en este caso Prometheus. Y en la parte derecha se ajusta la visualización: título, descripción, tipo de gráfico, colores, comportamiento, etcétera.

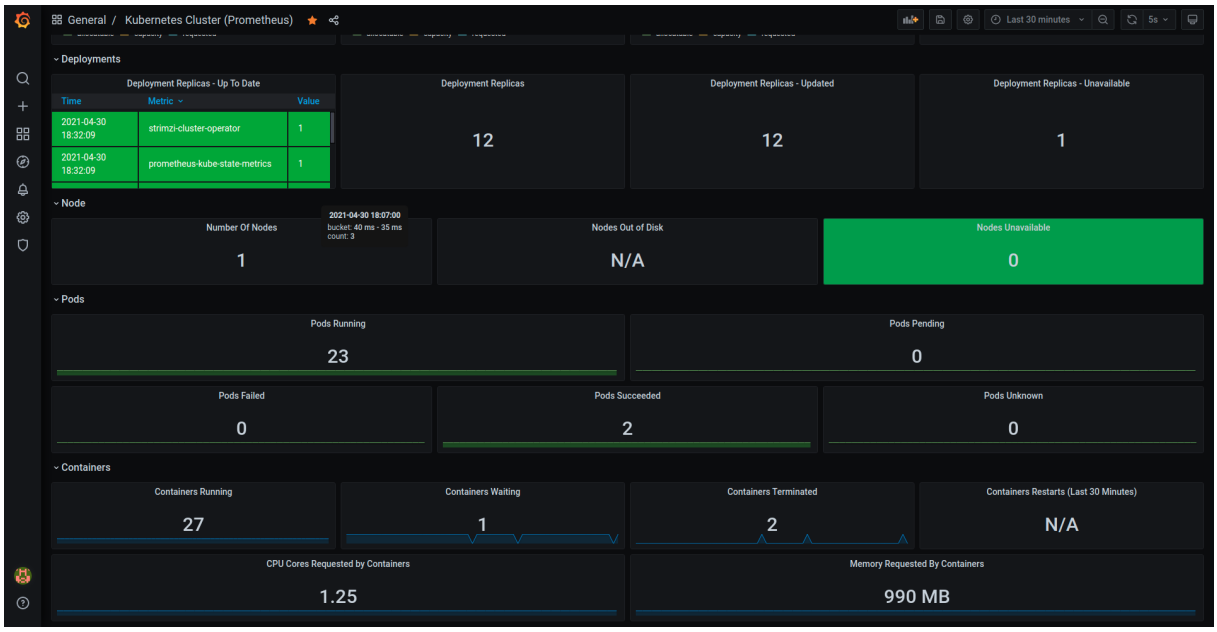


Figura 4.7: *Dashboard* con el resto de métricas del clúster de Kubernetes.

Figura 4.8: Configuración de un panel en Grafana.

4.8. Creación de un chart de Helm

Para simplificar el proceso de instalación de la plataforma y que sea replicable en cualquier proveedor cloud, se ha creado un *chart* de Helm. El repositorio que contiene el código desarrollado y el *chart* como tal puede encontrarse en https://gitlab.com/iok8s/charts.

4.8.1. Configuración básica del chart

Para crear un *chart* basta con ejecutar

```
helm create my-chart
```

sustituyendo `my-chart` con el nombre que se quiera. En este caso se utiliza `iok8s`. A continuación se generan en la carpeta donde se ejecutó el comando una serie de carpetas y archivos:

```
package-name/
  charts/
  templates/
  Chart.yaml
  values.yaml
```

Dentro de la carpeta `charts`, Helm descargará todos los *charts* de los que dependa el que se ha creado. En la carpeta `templates` se almacenan todos los ficheros de configuración que Helm debe desplegar. No importa si se separan en carpetas. De hecho, es recomendable hacerlo de esta forma para facilitar la comprensión. Los ficheros de configuración que se utilizan en este caso son los que se listan en la sección B y que se han referenciado en las secciones anteriores.

En el fichero `Chart.yaml` se definen los metadatos del *chart*: el nombre, la versión, otros *charts* de los que dependa junto con sus versiones, etcétera. Por otro lado, en `values.yaml` se establecen las variables que se van a utilizar y que cualquiera que instale el chart puede modificar. Estas variables se utilizan en las acciones de las plantillas de Go. Por ejemplo, en el listing B.7 se puede ver cómo se sustituye en la contraseña de conexión con Redis la variable `redisPassword` que se define en el Listing B.11. Cada usuario que instale este *chart* podrá utilizar su propia contraseña de conexión a Redis.

Una vez se completan estos ficheros y se incluyen en la carpeta `templates` todos los ficheros de configuración, se puede utilizar

```
helm install my-release .
```

para desplegar el *chart* en un clúster local. Para publicarlo y que cualquier persona pueda utilizarlo es necesario subirlo a un registro. En este caso se utiliza Gitlab y se aplica integración continua para automatizar la subida.

Integración continua

En el Listing B.12 se define la *pipeline* de integración continua para publicar el *chart* en el registro del repositorio de Gitlab. La dirección donde se publica es `registry.gitlab.com/iok8s/charts/iok8s:v0.1.0`. El primer `iok8s` se corresponde con el grupo de Gitlab que contiene todos los repositorios y el segundo con el nombre del *chart*. La versión, en este caso `v0.1.0`, es la versión del *tag* (etiqueta) de `git`. Solamente se publica una versión nueva del *chart* si se crea una nueva etiqueta.

Por otro lado, en la *pipeline* se añade un paso previo que se ejecuta incluso cuando no se publica un *tag*. Este paso ejecuta

```
helm lint iok8s
```

para comprobar que la sintaxis del *chart* sea la correcta y no genere problemas en la instalación.

4.8.2. Uso del chart

Para instalar el chart desde cualquier clúster de Kubernetes basta con ejecutar

```
export HELM_EXPERIMENTAL_OCI=1
helm chart pull registry.gitlab.com/iok8s/charts/iok8s:v0.1.0
helm chart export registry.gitlab.com/iok8s/charts/iok8s:v0.1.0 -d /tmp/
helm upgrade --install my-release /tmp/iok8s
```

Es necesario exportar la variable `HELM_EXPERIMENTAL_OCI` ya que la función `chart pull` de Helm aún está en un periodo experimental. Tras varios minutos todos los pods y servicios estarán disponibles. Se puede visualizar la creación de los recursos en tiempo real ejecutando:

```
watch -n 5 kubectl get pods
```

Tras unos minutos los recursos creados deberán ser los siguientes:

```
NAME                                                  READY   STATUS    RESTARTS   AGE
alertmanager-my-release-kube-prometheus-alertmanager-0   2/2     Running   0          4m51s
cluster-01-entity-operator-7998bf9bc4-dwhgs           3/3     Running   0          2m28s
cluster-01-kafka-0                                    1/1     Running   0          3m4s
cluster-01-kafka-exporter-79548f54dc-nrlsl            1/1     Running   0          99s
cluster-01-zookeeper-0                                1/1     Running   0          4m9s
collector-deployment-6b85d66dd9-qpmbd                 1/1     Running   4          5m37s
my-release-grafana-5c99cfc77-lnrjp                    2/2     Running   0          5m37s
my-release-kube-prometheus-operator-5bfdd6c86c-n7j72  1/1     Running   0          5m36s
my-release-kube-state-metrics-5ddbfbf6d6-qsts9        1/1     Running   0          5m37s
my-release-prometheus-node-exporter-8xkxc             1/1     Running   0          5m37s
prometheus-my-release-kube-prometheus-prometheus-0    2/2     Running   1          4m50s
redis-0                                               1/1     Running   0          5m36s
redis-1                                               1/1     Running   0          4m8s
redis-2                                               1/1     Running   0          3m52s
rule-engine-api-6b6f9bbff-qpbpp                       1/1     Running   2          5m37s
sentinel-0                                            1/1     Running   0          5m36s
sentinel-1                                            1/1     Running   0          3m21s
sentinel-2                                            1/1     Running   0          3m16s
strimzi-cluster-operator-cbb97bb58-f6bgv              1/1     Running   0          5m37s
```

Al instalar el *chart* se imprime por pantalla una breve ayuda para su uso que se puede ver a continuación:

```
Thank you for installing {{ .Chart.Name }}.

Your release is named {{ .Release.Name }}.

To learn more about the release, try:

  $ helm status {{ .Release.Name }}
  $ helm get all {{ .Release.Name }}

You can see how the pods are being created with

  $ watch -n 5 kubectl get pods

In case you are using Minikube, access the Grafana cluster using:

  $ minikube  service  {{ .Release.Name }}-grafana
```

De nuevo, las plantillas de Helm son de gran utilidad. Si se ha escogido como nombre de la *release* 'ejemplo', la ayuda se imprimirá conforme a ello. El `.Chart.Name` se sustituye por el nombre que se define en `Chart.yaml` (listing B.10).

4.9. Resumen del capítulo

En esta sección se ha explicado en qué consisten las herramientas que se han utilizado y porqué se han elegido. Se ha hecho especial incapié en Kubernetes debido a que es el núcleo y orquestador del resto de componentes, además una tecnología relativamente reciente. Es realmente útil disponer de herramientas tan potentes, de código libre y bien documentadas. Y no solo eso, operadores como el de Strimzi que permiten instalar este tipo de herramientas, escalaras y modificarlas de forma tan sencilla permite que en cuestión de días se puedan crear un producto mínimo viable o una prueba de concepto.

Se ha creado además un *chart* de Helm para simplificar la instalación y distribución de la plataforma.

Además, el desarrollo que se ha hecho no ha consumido muchos recursos: con 2GB de RAM y 2 núcleos de un CPU Intel i5-8250U ha servido. Dependiendo del tráfico entrante y el almacenamiento necesario habría que escalar las réplicas y crear volúmenes persistentes acordes, pero sería tan sencillo como modificar manualmente los ficheros de configuración o establecer autoescalado en Kubernetes.

Capítulo 5

Evaluación de la arquitectura

5.1. Introducción

En esta sección se presentan dos experimentos: el primero consiste en crear un consumidor sencillo en Python para visualizar los datos que se publican en la plataforma en tiempo real. El segundo comprueba cómo afecta en el rendimiento el escalado de los pods de la API y de Kafka.

5.2. Visualización en tiempo real

En esta primera prueba se utiliza el emulador de dispositivos que se presentó en la sección 4 para enviar mensajes aleatorios a la plataforma. Estos mensajes contienen una imagen de píxeles aleatorios en el campo `security camera`, un porcentaje de humedad en el campo `humidity` y una temperatura en `temperature`.

Para visualizar los datos en tiempo real se ha preparado un código muy simple en Python, que se encuentra en el Listing E.1. En él se representa la imagen del campo `security camera` junto con la temperatura y el porcentaje de humedad en la parte superior. En la parte inferior se indica el *topic*, la partición y el *offset*[1] del mensaje. Esta representación, que se puede ver en la Figura 5.1, se actualiza cada vez que llega un

[1] El *offset* es un identificador único que se le asigna a cada mensaje de forma incremental. El consumidor leerá los mensajes ordenados por el *offset*.

mensaje.

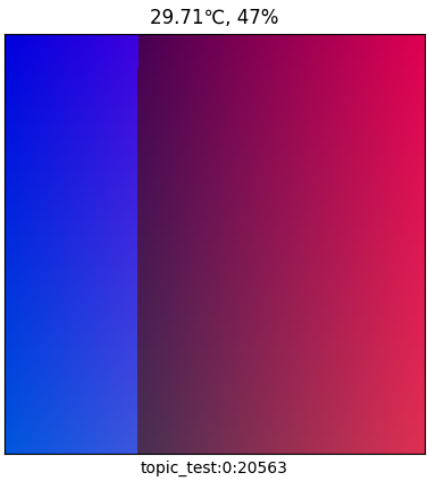


Figura 5.1: Fotograma de la visualización de los datos en tiempo real.

En la Figura 5.2 se representan los últimos 9 fotogramas sin leer para poder ver cómo el *offset* se incrementa y los distintos tipos de imágenes que se generan.

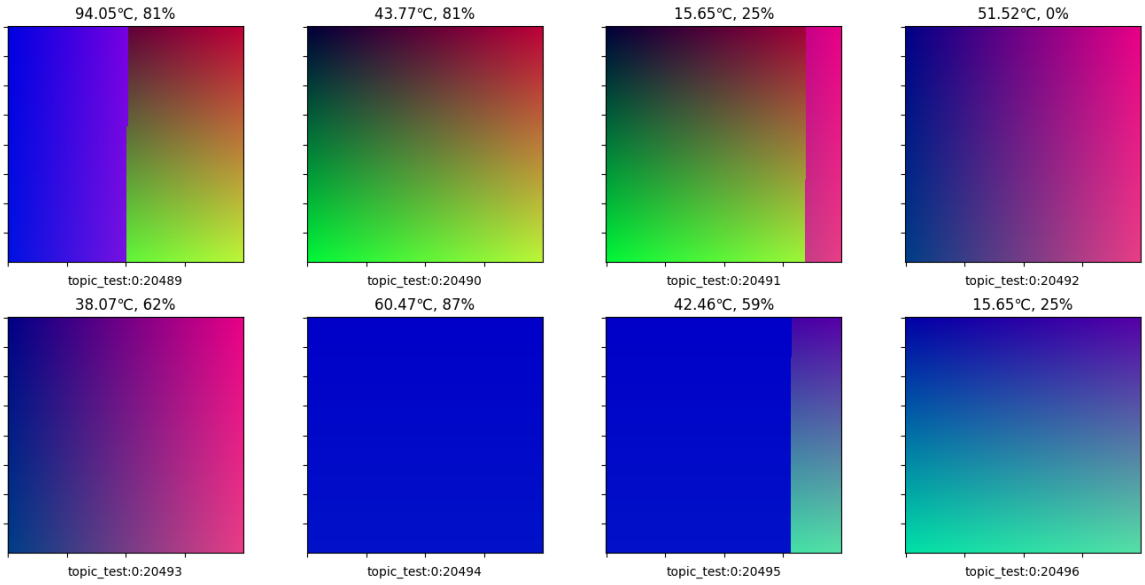


Figura 5.2: Representación de 9 mensajes recibidos en el consumidor.

Para poder conectar el consumidor al *broker* de Kafka se ha tenido que crear un *listener* en el Listing B.4 llamado *external*, que permite conexiones externas al clúster sin necesidad de utilizar certificados TLS.

5.3. Dependencia del rendimiento con el número de réplicas

5.3.1. Ejecución de la prueba de carga

En este segundo experimento se utiliza herramienta llamada `vegeta`[2] que permite realizar pruebas de carga sobre servidores HTTP. Se utiliza ésta en lugar del simulador de dispositivos porque genera un informe muy detallado con información sobre las peticiones: latencias, bytes enviados, recibidos y códigos de estado.

Lo que se busca con estas pruebas es comprobar si al aumentar las réplicas de la API y de los *brokers* de Kafka se consigue mejorar el rendimiento, ya que el escalado horizontal es uno de los motivos por los que se utiliza Kubernetes.

Para poder modificar el número de réplicas basta con editar las variables de Helm. Por ejemplo, para utilizar 2 réplicas:

```
helm upgrade --set collector.replicas=2 --set strimzi.kafkaReplicas=2 my-release /tmp/iok8s
```

Para ejecutar los tests se utiliza la siguiente *pipe* en `bash`:

```
echo "POST https://192.168.49.2:30500/collect" | vegeta attack -body data.json -duration 20s -insecure | tee results.bin | vegeta report
```

Primero se define la petición que se realiza, en este caso de tipo POST a la IP del clúster y al puerto expuesto por el servicio de la API. A continuación se inicia el 'ataque' de 20 segundos de duración, ignorando que el certificado TLS sea autofirmado y con un contenido en las peticiones que se lee desde el fichero `data.json`[3]. El último paso consiste en guardar los resultados en bruto en un fichero `results.bin` y convertirlo a un formato legible con `vegeta report`. El resultado será algo similar a:

```
Requests      [total, rate, throughput]  1000, 50.05, 44.72
Duration      [total, attack, wait]      19.99110999s, 19.979695246s, 11.414744ms
```

[2] https://github.com/tsenart/vegeta

[3] Este fichero contiene una petición generada por el simulador de dispositivos.

```
Latencies     [mean, 50, 95, 99, max]    461.970626ms, 412.088106ms,
   1.069372576s, 1.192038709s, 1.471774368s
Bytes In      [total, mean]              2862, 2.86
Bytes Out     [total, mean]              1058000, 1058.00
Success       [ratio]                    89.40%
Status Codes  [code:count]               202:894  500:106
Error Set:
500 Internal Server Error
```

En este ejemplo se puede ver que el ataque ha durado aproximadamente 20 segundos y se han emitido 1000 peticiones. Como resultado se han obtenido 894 códigos de respuesta de tipo 202 y 106 de tipo 500, es decir de error interno.

5.3.2. Resultados

Se ha realizado una serie de pruebas de carga para 1, 2, 3 y 4 réplicas tanto de la API como de los *brokers* de Kafka. Los resultados para la latencia y el porcentaje de errores en las peticiones se muestra en la tabla 5.1. Se han enviado 1000 peticiones en 20 segundos y se ha repetido la prueba 5 veces para calcular una media y desviación estándar de los resultados. Se ha realizado así para evitar tomar las medidas en momentos de mayor o menor carga del clúster.

Replicas	Latencia media (ms)	Latencia p50 (ms)	Latencia p95 (ms)	Latencia p99 (ms)	Latencia máxima (ms)
1	$8,35 \pm 4,69$	$5,67 \pm 0,97$	$12,22 \pm 12,21$	$68,29 \pm 166,55$	$155,26 \pm 233,19$
2	$7,56 \pm 8,55$	$5,25 \pm 0,80$	$11,64 \pm 30,78$	$58,69 \pm 286,52$	$136,69 \pm 398,62$
3	$8,70 \pm 8,13$	$5,43 \pm 1,26$	$16,03 \pm 53,72$	$105,05 \pm 173,12$	$173,85 \pm 225,89$
4	$9,93 \pm 14,50$	$6,11 \pm 45,10$	$28,50 \pm 61,14$	$115,29 \pm 378,54$	$201,73 \pm 401,28$

Tabla 5.1: Latencia media, percentil 50, 95 y 99, y máxima para distinto número de réplicas de la API y *brokers* de Kafka en el clúster de Kubernetes.

Se puede comprobar que no existe gran diferencia en la latencia media de las peticiones utilizando distinto número de réplicas. Sin embargo la desviación estándar aumenta con el número de réplicas, dándose el caso de que incluso supera al valor medio para 4 réplicas. Por otro lado, los percentiles 95 y 99, además de la latencia máxima, aumentan claramente con el número de réplicas.

Se puede ver también en la Figura 5.3 una representación de cuatro de estas pruebas en

el *dashboard* de Grafana. El gráfico de la izquierda representa la distribución de tiempos de procesamiento (eje vertical) en función de la hora (eje horizontal). Cada casilla comprende intervalos temporales de 5 ms para el tiempo de procesamiento y 1 minuto para el eje temporal. En la derecha se representa el número de mensajes por segundo que se registran en el clúster de Kafka, con el topic `topic-test` (el que se utiliza en las pruebas de carga) en azul.

Se aprecia que cuanto mayor latencia en la petición y mayor tiempo de procesado, menor será el pico de mensajes registrados en el clúster. Sin embargo, cada prueba envía 1000 peticiones así que el total del área de cada pico deberá ser la misma. Que el pico sea menor significa que se han procesado los mensajes en un mayor lapso de tiempo.

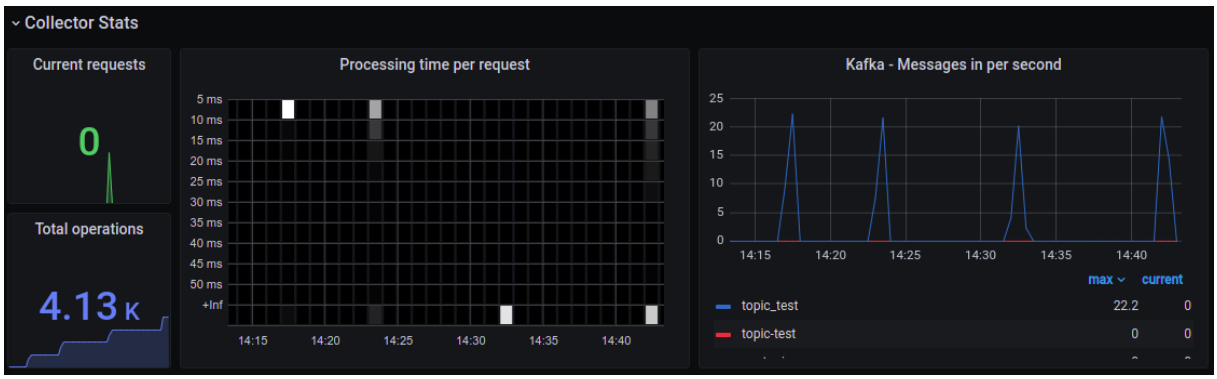


Figura 5.3: Paneles de Grafana para la monitorización de la API y el clúster de Kafka durante las pruebas de carga.

5.3.3. Análisis de resultados

Se comprueba que a medida que el número de réplicas aumenta, se incrementa con él la latencia de la petición y en una situación real debería disminuir o mantenerse constante en el peor de los casos. Existe una explicación y es que para el desarrollo de este trabajo se ha utilizado Minikube, que despliega un nodo máster y un nodo *worker* en local virtualizados con Docker. Al iniciar el clúster se reservan 2 CPUs y 2GB de memoria RAM, que se distribuyen entre todos los pods. Por lo tanto, al añadir más réplicas se reducen los recursos asignados a cada pod por individual. Además, se dedicarán parte de estos recursos a orquestar el correcto funcionamiento de todos los contenedores.

Por otro lado, cada pod tiene un consumo de recursos mínimo heredado del sistema

operativo que se ejecuta en el contenedor. Por ejemplo, en un despliegue donde un pod utiliza 50MB de memoria en reposo y le corresponden 250MB en total, tendrá 200MB restantes para procesar la petición entrante. Sin embargo, si en lugar de una sola réplica se utilizaran 4, se tendrían 200MB ocupados en reposo para un total de 250MB repartidos entre los 4 pods. Por lo tanto, solamente podrían utilizarse unos 12MB por pod para procesar las peticiones. Lo mismo ocurriría con las unidades de CPU.

Por lo tanto, para poder ver cómo se comportaría este despliegue al aumentar el número de réplicas habría que utilizar un clúster real. De esta forma se aseguraría un escalado horizontal real y los pods no verían sus recursos afectados.

5.4. Resumen

En el primer experimento se ha podido comprobar lo sencillo que resulta hacer integraciones sobre esta plataforma. En apenas 40 líneas de código se puede crear una interfaz gráfica para visualizar las imágenes de una cámara de seguridad o representar los cambios en las medidas de un sensor.

En la segunda prueba se comprueba que, al contrario de lo que se podría esperar, la latencia de las peticiones aumenta cuantas más réplicas se utilice. Sin embargo, no se puede concluir que esto sería lo que ocurriría en un clúster real de Kubernetes sino que es causa utilizar un entorno de desarrollo.

Capítulo 6

Conclusiones

En este último capítulo se repasan los objetivos que se plantearon y se valoran los resultados. Se finaliza con una opinión personal sobre lo aprendido y el posible trabajo futuro.

Objetivos A lo largo de la memoria se han resuelto los objetivos que se plantearon en un inicio:

- Se ha diseñado una arquitectura para una plataforma IoT basándose en bibliografía de otros estudios y productos comerciales.
- Se ha desarrollado el código necesario, que incluye la creación de una API de recolección de datos para el posterior almacenamiento y las configuraciones necesarias para desplegar un clúster de Kubernetes.

Considero que la solución es eficiente y escalable. Sin embargo, puede que la configuración del clúster no sea la adecuada o se hayan saltado algunas de las prácticas recomendadas. Por ejemplo, se podrían haber añadido límites para los recursos de cada pod. Este proyecto es mi primer contacto con la herramienta y puede que sea necesario revisar la configuración en busca de vulnerabilidades.

Por otro lado, al haberse desarrollado como prueba de concepto no se ha configurado ningún balanceador de carga, como podría ser Nginx, y la conexión con la API se hace mediante el puerto del nodo. Este detalle es lo primero que habría que cambiar si se fuera a desplegar en un clúster con acceso público o en un entorno *cloud*.

Finalmente, para poder empaquetar la aplicación y ser fácilmente replicable entre distintos grupos de trabajo, habría que crear un *chart* de Helm que contuviera todos los módulos de la plataforma y así poder desplegarlos con un solo comando.

Aprendizaje Diseñar e implementar la plataforma ha servido como un proyecto en el que aprender herramientas y nuevas formas de trabajar. Una vez diseñada la arquitectura me decidí a utilizar Kubernetes y Go a modo de reto, ya que nunca los había probado. Gracias a ello pude descubrir nuevas herramientas y aprender aún más.

Primero, aprendí Go mediante un curso muy completo (James, 2021) que utilizaba TDD (desarrollo guiado por pruebas) para explicar cada concepto. Era una metodología que conocía pero no acostumbraba a utilizar. Sin embargo, hoy en día, gracias a este trabajo, tanto en lo laboral como en proyectos personales trato de seguirla. El lenguaje como tal me ha parecido muy útil y sobretodo sencillo de utilizar. Existen además varios proyectos muy interesantes que utilizan este lenguaje (Terraform, Docker, Etcd) por lo que conocerlo va a permitirme participar en ellos.

Por otro lado, aprender a utilizar Kubernetes no fue sencillo y de hecho me quedan muchos detalles por investigar. Considero que ha sido una puerta de entrada al concepto *Cloud Native* y me ha permitido descubrir herramientas muy interesantes que me habría encantado incluir en el trabajo, como Jaegertracing o FluentD, y otras que he podido incorporar, como Prometheus.

Trabajo futuro Por supuesto, estudiar otras plataformas IoT me ha servido para conocer cómo otros diseñan sus sistemas y de qué forma conectan sus piezas. He podido explorar también servicios *cloud* que no conocía y descubrir los proyectos de código libre en los que se basan. A continuación, se enumeran algunas herramientas que se pueden añadir a la plataforma para darle aún más valor.

Ya sea para desarrollos sencillos u otros más complejos, la detección de errores en sistemas distribuidos es muy compleja si no se utilizan herramientas externas. Una opción que no implica modificar los servicios ni APIs que se hayan creado es el stack ELK[1] (ElasticSearch, Logstash y Kibana): un conjunto de productos de código abierto de la

[1] https://www.elastic.co/es/elastic-stack

empresa Elastic que permiten recoger dato con Logstash, almacenarlo en Elasticsearch y visualizarlo con Kibana. A partir de él surgen soluciones como EFK, que sustituye Logstash por FluentD[2] -un recolector de logs que se integra de forma muy sencilla con Kubernetes-, o incluso Loki[3], un proyecto similar a ELK pero de la empresa creadora de Grafana. Ya que se está utilizando Grafana para la visualización de métricas, lo lógico sería elegir Loki frente a ELK o EFK y así utilizar únicamente una herramienta de visualización.

Por otro lado, en lugar de almacenar los datos en los nodos de Kafka, se pueden utilizar conectores de Kafka para bases de datos como MongoDB[4] o sistemas de archivos como HDFS[5].

Como añadido al módulo de reglas, se podría utilizar Apache OpenWhisk[6], una herramienta FaaS (*Function as a Service*) que permite desplegar funciones *serverless* que se ejecutan en respuesta a eventos, por ejemplo de ingesta de datos. Permite programar las funciones en multitud de lenguajes y desplegarlas en Kubernetes. Además, incluye una librería para establecer Kafka como fuente de eventos[7].

Para el desarrollo de aplicaciones y microservicios se podría utilizar Jaeger[8], un proyecto de la CNCF escrito en Go que permite crear trazas en sistemas distribuidos de forma muy sencilla.

Por supuesto, si esta plataforma se quiere llevar a un entorno de producción debería utilizarse un clúster de Kubernetes de más de un nodo o bien un motor cloud como podría ser GKE (Google Kubernetes Engine) o Amazon EKS (Elastic Kubernetes Service). Además, habría que usar un proxy como Nginx[9] o Traefik[10], comprar un dominio y utilizar una autoridad certificadora válida como podría ser *Let's Encrypt* para renovar de forma automática los certificados[11].

Por otro lado, me gustaría crear una serie de conectores con los servicios más utili-

[2] https://www.fluentd.org/
[3] https://grafana.com/oss/loki/
[4] https://www.mongodb.com/kafka-connector
[5] https://github.com/confluentinc/kafka-connect-hdfs
[6] https://openwhisk.apache.org/
[7] https://github.com/apache/openwhisk-package-kafka
[8] https://www.jaegertracing.io/
[9] https://www.nginx.com/
[10] https://traefik.io/traefik/
[11] https://www.nginx.com/blog/using-free-ssltls-certificates-from-lets-encrypt-with-nginx/

zados como podrían ser Google Storage o AWS S3, BigQuery y GCP PubSub o AWS Kinesis, entre muchos otros. Además, para facilitar aún más la conexión con herramientas de Big Data, como Spark, se podría ofrecer una interfaz de Apache Zeppelin[12]. Éste permite ejecutar diferentes intérpretes, entre los que se incluyen Spark, SQL, Scala, Python, Cassandra y muchos más, en un *notebook* similar a los de Jupyter[13].

Creo que podría lograrse un producto completo y realmente útil que cualquier empresa o grupo de trabajo que quiera crear una plataforma IoT podría adoptar como núcleo.

Resumen En general estoy muy satisfecho con el trabajo realizado, aunque me habría gustado llegar un paso más allá y poder ofrecer un producto terminado que poder desplegar en la nube y utilizar directamente. Sin embargo, creo que con lo aprendido y con la base que he construido, voy a poder continuar con el proyecto e incluir todas aquellas piezas que he mencionado, y las que descubriré.

[12] https://zeppelin.apache.org/
[13] https://jupyter.org/

Bibliografía

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376.

Ashton, Kevin, e. a. (2009). That 'internet of things' thing. *RFID journal*, 22(7):97–114.

Augustin, A., Yi, J., Clausen, T., and Townsley, W. M. (2016). A study of lora: Long range & low power networks for the internet of things. *Sensors*, 16(9).

Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA.

Dickson, B. (2020). Iot botnets might be the cybersecurity industry's next big worry. https://www.iotsecurityfoundation.org/iot-botnets-might-be-the-cybersecurity-industrys-next-big-worry/. Último acceso: febrero de 2021.

Docs, S. (2021). Diagrama de microservicios. https://sitewhere.io/docs/2.1.0/es/platform/microservice-overview.html#estructura-de-microservicios. Último acceso: mayo de 2021.

Eclipse-Foundation (2019). Eclipse Hono™ iot platform. https://www.eclipse.org/hono/. Último acceso: marzo de 2021.

Emily Freeman, N. H. (2020). *97 Things Every Cloud Engineer Should Know*, pages 15–16.

Gerard Maas, F. G. (2019). *Stream Processing with Apache Spark*, chapter 21. O'Reilly Media, Inc.

Guth, J., Breitenbücher, U., Falkenthal, M., Fremantle, P., Kopp, O., Leymann, F., and Reinfurt, L. (2018). *A Detailed Analysis of IoT Platform Architectures: Concepts, Similarities, and Differences*, pages 86–96.

James, C. (2021). Learn go with tests. https://quii.gitbook.io/learn-go-with-tests/. Último acceso: abril de 2021.

Jerome Henry, R. B. (2017). Internet of things (iot) fundamentals [video]. https://www.oreilly.com/library/view/internet-of-things/9780134667577/. O'Reilly.

Kortuem, G., Kawsar, F., Sundramoorthy, V., and Fitton, D. (2010). Smart objects as building blocks for the internet of things. *IEEE Internet Computing*, 14(1):44–51.

Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. In Proc. of the 6th Intl. Workshop on Networking Meets Databases (NetDB). Athens, Greece.

Lopez-Araiza, C. and Cankaya, E. (2017). A comprehensive analysis of security tools for network forensics. *Journal of Medical - Clinical Research & Reviews*, 1:1–9.

Pastor-Vargas, R., Tobarra, L., Robles-Gómez, A., Martin, S., Hernández, R., and Cano, J. (2020). A wot platform for supporting full-cycle iot solutions from edge to cloud infrastructures: A practical case. *Sensors*, 20(13).

Patel, K. K., Patel, S. M., et al. (2016). Internet of things-iot: definition, characteristics, architecture, enabling technologies, application & future challenges. *International journal of engineering science and computing*, 6(5). International Journal of Engineering Science and Computing, Vol. 6, No. 5, pp.6123-6131.

Phiri, H. and Kunda, D. (2017). A comparative study of nosql and relational database. *Zambia ICT Journal*, 1:1–4.

Polato, I., Ré, R., Goldman, A., and Kon, F. (2014). A comprehensive view of hadoop research—a systematic literature review. *Journal of Network and Computer Applications*, 46:2.

Rayes, A. and Salam, S. (2017). *IoT Services Platform: Functions and Requirements*, pages 165–194. Capítulo del libro Internet of Things From Hype to Reality (pp.165-194).

repository, F. (2021). Helm charts. https://github.com/FIWARE/helm-charts/. Último acceso: mayo de 2021.

Ristevski, B. and Chen, M. (2018). Big data analytics in medicine and healthcare. *Journal of Integrative Bioinformatics*, 15.

Romkey, J. (2017). Toast of the iot: The 1990 interop internet toaster. *IEEE Consumer Electronics Magazine*, 6:116–119.

Singh, K., Behera, R., and Mantri, J. (2019). *Big Data Ecosystem: Review on Architectural Evolution: Proceedings of IEMIS 2018, Volume 2*, pages 335–345.

SiteWhere (2021). Guía de despliegue de sitewhere en kubernetes. https://sitewhere.io/docs/2.0.0-rc2/es/deployment/kubernetes.html. Último acceso: mayo de 2021.

Tamboli, A. (2019). *Build Your Own IoT Platform: Develop a Fully Flexible and Scalable Internet of Things Platform in 24 Hours*.

Tang, E. and Fan, Y. (2016). Performance comparison between five nosql databases. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 105–109.

ThingsBoard (2021). Open-source iot platform. thingsboard.io.

Traub, J., Grulich, P., Cuéllar, A., Breß, S., Katsifodimos, A., Rabl, T., and Markl, V. (2018). Scotty: Efficient window aggregation for out-of-order stream processing. In Proc. of IEEE 34th International Conference on Data Engineering (ICDE). Paris, France.

Tyler Akidau, Slava Chernyak, R. L. (2018). *Streaming Systems*. O'Reilly Media, Inc. https://www.oreilly.com/library/view/streaming-systems/9781491983867/.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA. USENIX Association.

Apéndice A

Lista de siglas, acrónimos y nuevos conceptos

AWS **A**mazon **W**eb **S**ervices es un proveedor de soluciones cloud.

Data Warehouse Almacén de grandes volúmenes de datos recogidos de fuentes diversas. A diferencia de otras formas de almacenamiento, cada organización tiene un *data warehouse* donde guardan los datos de todas sus aplicaciones.

ETL **E**xtract **T**ransform and **L**oad es una forma de procesar datos en Big Data. Se extraen de una fuente de datos, se transforman y se almacenan en la fuente destino.

FaaS **F**unction **a**s **a** **S**ervice es una categoría de la computación en la nube en la que se despliegan aplicaciones *serverless* que responden ante eventos. Por ejemplo, se podría programar una función que envír un correo cada vez que se sube un archivo a un SFTP y desplegarla en un entorno FaaS.

GCP **G**oogle **C**loud **P**latform es un proveedor de soluciones cloud.

IaaS **I**nfraestructure **a**s **a** **S**ervice o infraestructura como servicio. Se corresponde con los servicios en la nube que proveen de recursos para computación, almacenamiento, redes y seguridad, entre otros, en la nube.

IoT El **I**nternet **o**f **T**hings, o Internet de las Cosas, es un concepto que engloba a redes formadas mayormente por dispositivos conectados a Internet.

LDAP El **P**rotocolo **L**igero de **A**cceso a **D**irectorios permite gestionar usuarios, credenciales, permisos y recursos en redes corporativas.

El protocolo LDAP es muy utilizado actualmente por empresa que apuestan por el software libre al utilizar distribuciones de Linux para ejercer las funciones propias de un directorio activo en el que se gestionarán las credenciales y permisos de los trabajadores y estaciones de trabajo en redes LAN corporativas en conexiones cliente/servidor.

MapReduce (Dean and Ghemawat, 2004). Sistema de procesamiento de datos distribuido y paralelo utilizado en Hadoop. Las tareas MapReduce se componen de dos funciones o etapas. *Map* recibe un archivo de entrada y devuelve una lista de pares de la forma (clave, valor). *Reduce* recibe como entrada las listas resultado de la función *map* en cada uno de los nodos y las combina para cada clave.

Un ejemplo podría ser un contador de palabras de un libro donde cada nodo del clúster se encarga de aplicar *map* a una página. El resultado de la función sería una lista de la forma `(palabra, 1)`, pudiendo haber palabras repetidas. *Reduce* se encargaría de sumar el valor para cada palabra, en este caso 1 para cada vez que apareciera, y devolver como resultado una lista de la forma `(palabra, valor)`, siendo el valor el número de apariciones en el libro.

Proveedor de soluciones cloud Permiten disponer de aplicaciones, herramientas preinstaladas y servidores en los centros de procesamiento de los proveedores para evitar que cada empresa deba alojar, gestionar e instalar las aplicaciones más comunes en infraestructura propia. Engloban desde soluciones informáticas para desarrolladores (instancias cloud, clústeres para procesamiento distribuido, *data warehouse*, modelos de aprendizaje automático, seguridad informática, etc.) hasta soluciones en otros sectores como por ejemplo publicidad y marketing, telecomunicaciones o banca.

NoSQL **N**ot only **SQL** son un tipo de bases de datos no relacional. Comprenden tipos como bases de datos de clave valor, de grafos, de documentos u orientadas a columnas.

SQL **S**tructured **Q**uery **L**anguage es un lenguaje de consultas para bases de datos relacionales.

TTM **T**ime **T**o **M**arket. Periodo de tiempo desde que una empresa decide comenzar un proyecto hasta que está listo para su comercialización.

TTD **T**est **D**riven **D**evelopment, o desarrollo basado en pruebas, es una metodología de trabajo ágil que consiste en crear pruebas del código antes de desarrollar las funcionalidades.

Apéndice B

Ficheros de configuración

Todos los ficheros se han adecuado para poder utilizarse con Helm. Por ese motivo se pueden encontrar algunos valores sustituidos por variables entre corchetes, propio de las plantillas de Go.

B.1. API de recolección

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: collector-deployment
  labels:
    app: collector
spec:
  replicas: {{ .Values.collector.replicas}}
  selector:
    matchLabels:
      app: collector
  template:
    metadata:
      labels:
        app: collector
    spec:
      containers:
      - name: collector
```

```
19          image: juandspy/iok8s-collector:v0.5
20          imagePullPolicy: Always
21          ports:
22          - containerPort: 5000
23          args: ["-kafkaAddress={{ .Values.strimzi.clusterName }}-
    kafka-bootstrap:9092", "-ruleEngineUrl=http://rule-engine-api-
    service:3000"]
```

Listing B.1: Despliegue de la API de recolección. Parte del fichero `collector.yaml`

```yaml
apiVersion: v1
kind: Service
metadata:
  name: collector-service
  labels:
    app: collector-service
spec:
  type: NodePort
  selector:
    app: collector
  ports:
    - name: https
      protocol: TCP
      port: 5000
      targetPort: 5000
      nodePort: {{ .Values.collector.service.nodePort }}
```

Listing B.2: Servicio para la API de recolección. Parte del fichero `collector.yaml`.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: collector-metrics-servicemonitor
  labels:
    app: collector
    release: {{ $.Release.Name | quote }}
spec:
  selector:
    matchLabels:
      app: collector-service
  endpoints:
  - port: https
    interval: 15s
    tlsConfig:
      insecureSkipVerify: true
    scheme: https
```

Listing B.3: Monitorización del servicio para la API de recolección. Parte del fichero `collector.yaml`.

B.2. Clúster de Kafka

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: {{ .Values.strimzi.clusterName }}
spec:
  kafka:
    version: 2.7.0
    replicas: {{ .Values.strimzi.kafkaReplicas }}
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
      - name: external
        port: 9094
        type: nodeport
        tls: false
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min.isr: 1
      log.message.format.version: "2.7"
      inter.broker.protocol.version: "2.7"
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 5Gi
        deleteClaim: false
    metricsConfig:
      type: jmxPrometheusExporter
```

```
37        valueFrom:
38          configMapKeyRef:
39            name: kafka-metrics
40            key: kafka-metrics-config.yml
41
42    zookeeper:
43      replicas: {{ .Values.strimzi.zookeeperReplicas }}
44      storage:
45        type: persistent-claim
46        size: 5Gi
47        deleteClaim: false
48      metricsConfig:
49        type: jmxPrometheusExporter
50        valueFrom:
51          configMapKeyRef:
52            name: kafka-metrics
53            key: zookeeper-metrics-config.yml
54
55    entityOperator:
56      topicOperator: {}
57      userOperator: {}
58
59    kafkaExporter:
60      topicRegex: ".*"
61      groupRegex: ".*"
62  ---
63  kind: ConfigMap
64  apiVersion: v1
65  metadata:
66    name: kafka-metrics
67    labels:
68      app: strimzi
69  data:
70    kafka-metrics-config.yml: |
71      # See https://github.com/prometheus/jmx_exporter for more
     info about JMX Prometheus Exporter metrics
72      lowercaseOutputName: true
73      rules:
74      # Special cases and very specific rules
```

```
75    - pattern: kafka.server<type=(.+), name=(.+), clientId=(.+),
      topic=(.+), partition=(.*)><>Value
76      name: kafka_server_$1_$2
77      type: GAUGE
78      labels:
79       clientId: "$3"
80       topic: "$4"
81       partition: "$5"
82    - pattern: kafka.server<type=(.+), name=(.+), clientId=(.+),
      brokerHost=(.+), brokerPort=(.+)><>Value
83      name: kafka_server_$1_$2
84      type: GAUGE
85      labels:
86       clientId: "$3"
87       broker: "$4:$5"
88    - pattern: kafka.server<type=(.+), cipher=(.+), protocol=(.+),
      listener=(.+), networkProcessor=(.+)><>connections
89      name: kafka_server_$1_connections_tls_info
90      type: GAUGE
91      labels:
92        listener: "$2"
93        networkProcessor: "$3"
94        protocol: "$4"
95        cipher: "$5"
96    - pattern: kafka.server<type=(.+), clientSoftwareName=(.+),
      clientSoftwareVersion=(.+), listener=(.+), networkProcessor
      =(.+)><>connections
97      name: kafka_server_$1_connections_software
98      type: GAUGE
99      labels:
100       clientSoftwareName: "$2"
101       clientSoftwareVersion: "$3"
102       listener: "$4"
103       networkProcessor: "$5"
104   - pattern: "kafka.server<type=(.+), listener=(.+),
      networkProcessor=(.+)><>(.+):"
105     name: kafka_server_$1_$4
106     type: GAUGE
107     labels:
```

```
108         listener: "$2"
109         networkProcessor: "$3"
110       - pattern: kafka.server<type=(.+), listener=(.+),
    networkProcessor=(.+)><>(.+)
111         name: kafka_server_$1_$4
112         type: GAUGE
113         labels:
114           listener: "$2"
115           networkProcessor: "$3"
116       # Some percent metrics use MeanRate attribute
117       # Ex) kafka.server<type=(KafkaRequestHandlerPool), name=(
    RequestHandlerAvgIdlePercent)><>MeanRate
118       - pattern: kafka.(\w+)<type=(.+), name=(.+)Percent\w*><>
    MeanRate
119         name: kafka_$1_$2_$3_percent
120         type: GAUGE
121       # Generic gauges for percents
122       - pattern: kafka.(\w+)<type=(.+), name=(.+)Percent\w*><>Value
123         name: kafka_$1_$2_$3_percent
124         type: GAUGE
125       - pattern: kafka.(\w+)<type=(.+), name=(.+)Percent\w*, (.+)
    =(.+)><>Value
126         name: kafka_$1_$2_$3_percent
127         type: GAUGE
128         labels:
129           "$4": "$5"
130       # Generic per-second counters with 0-2 key/value pairs
131       - pattern: kafka.(\w+)<type=(.+), name=(.+)PerSec\w*, (.+)
    =(.+), (.+)=(.+)><>Count
132         name: kafka_$1_$2_$3_total
133         type: COUNTER
134         labels:
135           "$4": "$5"
136           "$6": "$7"
137       - pattern: kafka.(\w+)<type=(.+), name=(.+)PerSec\w*, (.+)
    =(.+)><>Count
138         name: kafka_$1_$2_$3_total
139         type: COUNTER
140         labels:
```

```
      "$4": "$5"
  - pattern: kafka.(\w+)<type=(.+), name=(.+)PerSec\w*><>Count
    name: kafka_$1_$2_$3_total
    type: COUNTER
  # Generic gauges with 0-2 key/value pairs
  - pattern: kafka.(\w+)<type=(.+), name=(.+), (.+)=(.+), (.+)
=(.+)><>Value
    name: kafka_$1_$2_$3
    type: GAUGE
    labels:
      "$4": "$5"
      "$6": "$7"
  - pattern: kafka.(\w+)<type=(.+), name=(.+), (.+)=(.+)><>Value
    name: kafka_$1_$2_$3
    type: GAUGE
    labels:
      "$4": "$5"
  - pattern: kafka.(\w+)<type=(.+), name=(.+)><>Value
    name: kafka_$1_$2_$3
    type: GAUGE
  # Emulate Prometheus 'Summary' metrics for the exported '
Histogram's.
  # Note that these are missing the '_sum' metric!
  - pattern: kafka.(\w+)<type=(.+), name=(.+), (.+)=(.+), (.+)
=(.+)><>Count
    name: kafka_$1_$2_$3_count
    type: COUNTER
    labels:
      "$4": "$5"
      "$6": "$7"
  - pattern: kafka.(\w+)<type=(.+), name=(.+), (.+)=(.*), (.+)
=(.+)><>(\d+)thPercentile
    name: kafka_$1_$2_$3
    type: GAUGE
    labels:
      "$4": "$5"
      "$6": "$7"
      quantile: "0.$8"
  - pattern: kafka.(\w+)<type=(.+), name=(.+), (.+)=(.+)><>Count
```

```
176       name: kafka_$1_$2_$3_count
177       type: COUNTER
178       labels:
179         "$4": "$5"
180     - pattern: kafka.(\w+)<type=(.+), name=(.+), (.+)=(.*)><>(\d+)
    thPercentile
181       name: kafka_$1_$2_$3
182       type: GAUGE
183       labels:
184         "$4": "$5"
185         quantile: "0.$6"
186     - pattern: kafka.(\w+)<type=(.+), name=(.+)><>Count
187       name: kafka_$1_$2_$3_count
188       type: COUNTER
189     - pattern: kafka.(\w+)<type=(.+), name=(.+)><>(\d+)
    thPercentile
190       name: kafka_$1_$2_$3
191       type: GAUGE
192       labels:
193         quantile: "0.$4"
194   zookeeper-metrics-config.yml: |
195     # See https://github.com/prometheus/jmx_exporter for more
    info about JMX Prometheus Exporter metrics
196     lowercaseOutputName: true
197     rules:
198     # replicated Zookeeper
199     - pattern: "org.apache.ZooKeeperService<name0=
    ReplicatedServer_id(\\d+)><>(\\w+)"
200       name: "zookeeper_$2"
201       type: GAUGE
202     - pattern: "org.apache.ZooKeeperService<name0=
    ReplicatedServer_id(\\d+), name1=replica.(\\d+)><>(\\w+)"
203       name: "zookeeper_$3"
204       type: GAUGE
205       labels:
206         replicaId: "$2"
207     - pattern: "org.apache.ZooKeeperService<name0=
    ReplicatedServer_id(\\d+), name1=replica.(\\d+), name2=(\\w+)><
    >(Packets\\w+)"
```

```
208       name: "zookeeper_$4"
209       type: COUNTER
210       labels:
211         replicaId: "$2"
212         memberType: "$3"
213     - pattern: "org.apache.ZooKeeperService<name0=
      ReplicatedServer_id(\\d+), name1=replica.(\\d+), name2=(\\w+)><
      >(\\w+)"
214       name: "zookeeper_$4"
215       type: GAUGE
216       labels:
217         replicaId: "$2"
218         memberType: "$3"
219     - pattern: "org.apache.ZooKeeperService<name0=
      ReplicatedServer_id(\\d+), name1=replica.(\\d+), name2=(\\w+),
      name3=(\\w+)><>(\\w+)"
220       name: "zookeeper_$4_$5"
221       type: GAUGE
222       labels:
223         replicaId: "$2"
224         memberType: "$3"
```

Listing B.4: Fichero `kafka-persistent.yaml` con el despliegue de los componentes del clúster de Kafka.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kafka-exporter-service
  labels:
    app: kafka-exporter-service
spec:
  type: NodePort
  selector:
    strimzi.io/name: {{ .Values.strimzi.clusterName }}-kafka-
    exporter
  ports:
    - name: metrics
      protocol: TCP
      port: 9404
      targetPort: 9404
```

Listing B.5: Primera parte del fichero `kafka-exporter.yaml` con el servicio para el pod de Kafka-exporter.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: kafka-exporter-metrics-servicemonitor
  labels:
    app: kafka-exporter-service
    release: {{ $.Release.Name | quote }}
spec:
  selector:
    matchLabels:
      app: kafka-exporter-service
  endpoints:
  - port: metrics
    interval: 15s
    scheme: http
```

Listing B.6: Segunda parte del fichero `kafka-exporter.yaml` con la monitorización para el servicio de Kafka-exporter.

B.3. Módulo de reglas

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rule-engine-api
  labels:
    role: rule-engine
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rule-engine-api
  template:
    metadata:
      labels:
        app: rule-engine-api
    spec:
      containers:
      - name: rule-engine-api
        image: juandspy/iok8s-rule_engine:v0.4
        imagePullPolicy: Always
        ports:
        - containerPort: 3000
        env:
          - name: REDIS_HOST
            value: redis      # redis service
          - name: REDIS_PORT
            value: "6379"
          - name: REDIS_PASSWORD
            value: {{ .Values.ruleEngine.redisPassword | quote }}
---
apiVersion: v1
kind: Service
metadata:
  name: rule-engine-api-service
  labels:
    role: rule-engine
```

```
spec:
  type: NodePort
  selector:
    app: rule-engine-api
  ports:
    - name: http
      protocol: TCP
      port: 3000
      targetPort: 3000
      nodePort: {{ .Values.ruleEngine.service.nodePort }}
```

Listing B.7: Fichero `rest-api.yaml` con el despliegue de la API del módulo de reglas, junto con su servicio.

El *configmap* de Redis no se adjunta por su extensión. Las líneas más relevantes del archivo son las siguientes:

```
masterauth {{ .Values.ruleEngine.redisPassword }}
requirepass {{ .Values.ruleEngine.redisPassword }}
```

La primera define la contraseña de acceso desde API al máster y requirepass la comunicación worker-máster. Se invita a consultar en:

https://gitlab.com/iok8s/charts/-/blob/master/iok8s/templates/rule_engine/redis-configmap.yaml

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
  labels:
    role: rule-engine
spec:
  serviceName: redis
  replicas: {{ .Values.ruleEngine.redisReplicas}}
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      initContainers:
      - name: config
        image: redis:6.2.3-alpine
        command: [ "sh", "-c" ]
        args:
          - |
            cp /tmp/redis/redis.conf /etc/redis/redis.conf

            echo "finding master..."
            MASTER_FDQN=`hostname  -f | sed -e 's/redis-[0-9]\./redis-0./'`
            if [ "$(redis-cli -h sentinel -p 5000 ping)" != "PONG" ]; then
              echo "master not found, defaulting to redis-0"

              if [ "$(hostname)" == "redis-0" ]; then
                echo "this is redis-0, not updating config..."
              else
                echo "updating redis.conf..."
                echo "slaveof $MASTER_FDQN 6379" >> /etc/redis/redis.conf
              fi
```

```
            else
              echo "sentinel found, finding master"
              MASTER="$(redis-cli -h sentinel -p 5000 sentinel
    get-master-addr-by-name mymaster | grep -E
    '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}')"
              echo "master found : $MASTER, updating redis.conf"
              echo "slaveof $MASTER 6379" >> /etc/redis/redis.
    conf
            fi
        volumeMounts:
        - name: redis-config
          mountPath: /etc/redis/
        - name: config
          mountPath: /tmp/redis/
      containers:
      - name: redis
        image: redis:6.2.3-alpine
        command: ["redis-server"]
        args: ["/etc/redis/redis.conf"]
        ports:
        - containerPort: 6379
          name: redis
        volumeMounts:
        - name: data
          mountPath: /data
        - name: redis-config
          mountPath: /etc/redis/
      volumes:
      - name: redis-config
        emptyDir: {}
      - name: config
        configMap:
          name: redis-config
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "standard"
```

```
      resources:
        requests:
          storage: 50Mi
---
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    role: rule-engine
spec:
  clusterIP: None
  ports:
  - port: 6379
    targetPort: 6379
    name: redis
  selector:
    app: redis
```

Listing B.8: Fichero `redis_statefulset.yaml` con el despliegue de los nodos trabajadores de Redis.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sentinel
  labels:
    role: rule-engine
spec:
  serviceName: sentinel
  replicas: {{ .Values.ruleEngine.sentinelReplicas}}
  selector:
    matchLabels:
      app: sentinel
  template:
    metadata:
      labels:
        app: sentinel
    spec:
      initContainers:
      - name: config
        image: redis:6.2.3-alpine
        command: [ "sh", "-c" ]
        args:
          - |
            REDIS_PASSWORD={{ .Values.ruleEngine.redisPassword }}

            echo "Waiting for redis service to be up"
            CONN=`(printf "AUTH $REDIS_PASSWORD\r\n";) | nc redis
    6379 | grep OK`
            while [ -z "$CONN" ]; do
                sleep 10
                echo "Retry Redis ping... "
                CONN=`(printf "AUTH $REDIS_PASSWORD\r\n";) | nc
    redis 6379 | grep OK`
                echo $CONN
            done
            echo "Redis service is up"

            {{- $nodes := ""}}
            {{- $namespace := .Release.Namespace }}
```

```
38          {{- range $index := until ( .Values.ruleEngine.
    redisReplicas | int ) }}
39            {{- $nodes = (printf "%v,redis-%v.redis.%v.svc.
    cluster.local" $nodes $index $namespace ) }}
40          {{- end }}
41          nodes={{ trimAll "," $nodes }}
42
43          MASTER=""
44          while [ "$MASTER" == "" ]
45          do
46            for i in ${nodes//,/ }
47            do
48                echo "finding master at $i"
49                MASTER=$(redis-cli --no-auth-warning --raw -h
    $i -a $REDIS_PASSWORD info replication | awk '{print $1}' |
    grep master_host: | cut -d ":" -f2)
50                if [ "$MASTER" == "" ]; then
51                    echo "no master found"
52                    MASTER=""
53                else
54                    echo "found $MASTER"
55                    break
56                fi
57            done
58          done
59
60          echo "sentinel monitor mymaster $MASTER 6379 2" >> /
    tmp/master
61          echo "port 5000
62          sentinel resolve-hostnames yes
63          sentinel announce-hostnames yes
64          $(cat /tmp/master)
65          sentinel down-after-milliseconds mymaster 5000
66          sentinel failover-timeout mymaster 60000
67          sentinel parallel-syncs mymaster 1
68          sentinel auth-pass mymaster $REDIS_PASSWORD
69          " > /etc/redis/sentinel.conf
70          cat /etc/redis/sentinel.conf
71      volumeMounts:
```

```
        - name: redis-config
          mountPath: /etc/redis/
      containers:
      - name: sentinel
        image: redis:6.2.3-alpine
        command: ["redis-sentinel"]
        args: ["/etc/redis/sentinel.conf"]
        ports:
        - containerPort: 5000
          name: sentinel
        volumeMounts:
        - name: redis-config
          mountPath: /etc/redis/
        - name: data
          mountPath: /data
      volumes:
      - name: redis-config
        emptyDir: {}
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "standard"
      resources:
        requests:
          storage: 50Mi
---
apiVersion: v1
kind: Service
metadata:
  name: sentinel
  labels:
    role: rule-engine
spec:
  clusterIP: None
  ports:
  - port: 5000
    targetPort: 5000
```

```
111     name: sentinel
112   selector:
113     app: sentinel
```

Listing B.9: Fichero `sentinel_statefulset.yaml` con el despliegue de los nodos Sentinel de Redis.

B.4. Chart de Helm

```yaml
apiVersion: v2
name: iok8s
description: A Helm chart for creating an Internet of Kubernetes cluster

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be packaged into versioned archives
# to be deployed.
#
# Library charts provide useful utilities or functions for the chart developer. They're included as
# a dependency of application charts to inject those utilities and functions into the rendering
# pipeline. Library charts do not define any templates and therefore cannot be deployed.
type: application

# This is the chart version. This version number should be incremented each time you make changes
# to the chart and its templates, including the app version.
# Versions are expected to follow Semantic Versioning (https://semver.org/)
version: 0.1.0

# This is the version number of the application being deployed. This version number should be
# incremented each time you make changes to the application. Versions are not expected to
# follow Semantic Versioning. They should reflect the version the application is using.
# It is recommended to use it with quotes.
appVersion: "0.1.0"

dependencies:
```

```yaml
- name: kube-prometheus-stack
  version: "16.1.2"
  repository: https://prometheus-community.github.io/helm-charts
- name: strimzi-kafka-operator
  version: "0.23.0"
  repository: https://strimzi.io/charts/
```

Listing B.10: Fichero `Chart.yaml` con la definición del *chart* de Helm.

```yaml
collector:
  replicas: 1
  service:
    nodePort: 30500
ruleEngine:
  redisPassword: "secret-password"
  redisReplicas: 3
  sentinelReplicas: 3
  service:
    nodePort: 30300
strimzi:
  clusterName: "cluster-01"
  kafkaReplicas: 1
  zookeeperReplicas: 1
```

Listing B.11: Fichero `values.yaml` con la definición de las variables del *chart*.

```yaml
image:
  name: alpine/helm:3.2.1
  entrypoint: ["/bin/sh", "-c"]

variables:
  HELM_EXPERIMENTAL_OCI: 1

stages:
  - lint-helm-chart
  - release-helm-chart

lint-helm:
  stage: lint-helm-chart
  script:
    - helm lint iok8s

release-helm:
  stage: release-helm-chart
  script:
    - echo "using user $CI_REGISTRY_USER"
    - helm registry login -u $CI_REGISTRY_USER -p
   $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - helm chart save iok8s $CI_REGISTRY/iok8s/charts/iok8s
   :$CI_COMMIT_TAG
    - helm chart push $CI_REGISTRY/iok8s/charts/iok8s
   :$CI_COMMIT_TAG
  only:
    - tags
```

Listing B.12: Fichero `.gitlab-ci.yml` con la definición de la*pipeline*.

Apéndice C

Código de la API REST de recolección

C.1. Emulador

```
package emulator

import (
  "bytes"
  "encoding/base64"
  "encoding/json"
  "image"
  "image/color"
  "image/png"
  "math/rand"
  "strconv"
  "time"

  "github.com/Szeliga/goray/engine" // image creation
  "github.com/segmentio/ksuid"      // uuid generation
)

type Device struct {
  Id      string
  sources map[string]string
```

```
}

func NewGeneralDevice() Device {
  return Device{
    Id: ksuid.New().String(),
    sources: map[string]string{
      "humidity":        "int",
      "temperature":     "float",
      "security camera": "image",
    },
  }
}

const randomImageWidth = 200
const randomImageHeight = 200
const floatPrecission = 4

type result struct {
  fieldName  string
  fieldValue string
}

func (d *Device) Measure() map[string]string {
  measure := make(map[string]string)
  measureChannel := make(chan result)

  for sensor, dataType := range d.sources {
    go func(s, d string) {
      measureChannel <- result{s, getRNGFromType(d)()}
    }(sensor, dataType)
  }

  for i := 0; i < len(d.sources); i++ {
    m := <-measureChannel
    measure[m.fieldName] = m.fieldValue
  }

  return measure
}
```

```
60
61 func (d *Device) GeneratePayload(measure map[string]string) ([]byte,
       error) {
62   payloadAsMap := map[string]interface{}{
63     "id":   d.Id,
64     "data": measure,
65   }
66   return json.Marshal(payloadAsMap)
67 }
68
69 func getRNGFromType(dataType string) func() string {
70   switch dataType {
71   case "int":
72     return GenerateRandomInt
73   case "float":
74     return GenerateRandomFloat
75   case "image":
76     return GenerateRandomImage
77   }
78   return nil
79 }
80
81 func GenerateRandomInt() string {
82   n := rand.Intn(100)
83   return strconv.Itoa(n)
84 }
85
86 func GenerateRandomFloat() string {
87   f := rand.Float64() * 100
88   return strconv.FormatFloat(f, 'f', floatPrecission, 64)
89 }
90
91 func GenerateRandomImage() string {
92   scene := engine.NewScene(randomImageWidth, randomImageHeight)
93   scene.EachPixel(func(x, y int) color.RGBA { return randomColor(x, y)
       })
94
95   imgBase64Str := imageToBase64(scene.Img)
96   return "" + imgBase64Str
```

```
97 }
98
99 func randomColor(x, y int) color.RGBA {
100   rand := rand.New(rand.NewSource(time.Now().Unix()))
101   return color.RGBA{
102     uint8(x * rand.Intn(255) / randomImageWidth),
103     uint8(y * rand.Intn(255) / randomImageHeight),
104     uint8(rand.Intn(255)),
105     255}
106 }
107
108 func imageToBase64(img image.Image) string {
109   var buf bytes.Buffer
110   png.Encode(&buf, img)
111   return base64.StdEncoding.EncodeToString(buf.Bytes())
112 }
```

Listing C.1: Fichero `emulator.go` en el que se define un simulador de dispositivos capaz de generar una imagen con píxeles aleatorios, una temperatura y un porcentaje de humedad.

```go
package emulator

import (
  "math/rand"
  "testing"
)

var randomSeed = int64(4)

func init() {
  rand.Seed(randomSeed)
}

var device = NewGeneralDevice()

var fieldsToCheck = []string{"humidity", "temperature", "security camera
    "}

func TestDeviceMeasure(t *testing.T) {
  gotMeasure := device.Measure()

  for _, field := range fieldsToCheck {
    assertKeyAndValueInMap(t, gotMeasure, field)
  }
}

func TestDeviceGetPayload(t *testing.T) {
  measure := map[string]string{
    "humidity":        GenerateRandomInt(),
    "temperature":     GenerateRandomFloat(),
    "security camera": GenerateRandomImage(),
  }

  _, err := device.GeneratePayload(measure)
  if err != nil {
    t.Errorf("Error generating the payload, %v", err)
  }
}

```

```
39 func assertKeyAndValueInMap(t *testing.T, m map[string]string, key
       string) {
40   value, exists := m[key]
41   if !exists {
42     t.Errorf("wanted %v as key in %v", key, m)
43   }
44
45   if value == "" {
46     t.Errorf("value of %v is empty", key)
47   }
48
49 }
```

Listing C.2: Fichero `emulator_test.go` en el que se someten a test los diferentes componentes de `emulator.go`.

C.2. Colector

```
package server

import (
  "encoding/json"
  "io/ioutil"
  "net/http"

  "github.com/prometheus/client_golang/prometheus"
  "github.com/prometheus/client_golang/prometheus/promauto"
  "github.com/prometheus/client_golang/prometheus/promhttp"
  log "github.com/sirupsen/logrus"
)

// Publisher implements a Publish function that returns an error and a
   Destoy function that closes the possible connections it may have
type Publisher interface {
  Publish(id string, payload interface{}) error
  Destroy() error
}

type CollectorServer struct {
  Pub Publisher
}

type DataJSON map[string]json.RawMessage

const WrongMethodErrorMsg = "method not allowed"
const EmptyBodyMsg = "the request body was empty or unprocessable"
const CannotDecodeErrorMsg = "cannot decode request body into json"
const CannotFindDataErrorMsg = "cannot find a data field in the request
   body or is empty"
const CannotFindIdErrorMsg = "cannot find an id field in the request
   body or is empty"
const CannotPublishMessage = "cannot publish the message"

func (c *CollectorServer) ServeHTTP(w http.ResponseWriter, r *http.
```

```
    Request) {
34    router := http.NewServeMux()
35    router.Handle("/collect", http.HandlerFunc(c.dataHandler))
36    router.Handle("/metrics", promhttp.Handler())
37
38    router.ServeHTTP(w, r)
39  }
40
41  var (
42    opsProcessed = promauto.NewCounter(prometheus.CounterOpts{
43      Name: "collector_processed_ops_total",
44      Help: "The total number of processed messages",
45    })
46    opsCurrent = promauto.NewGauge(prometheus.GaugeOpts{
47      Name: "collector_current_ops",
48      Help: "The current number of requests",
49    })
50    processingTime = prometheus.NewHistogram(prometheus.HistogramOpts{
51      Name:    "collector_processing_time_seconds",
52      Help:    "The processing time of messages in seconds.",
53      Buckets: prometheus.LinearBuckets(0.005, 0.005, 10),
54    })
55  )
56
57  func init() {
58    prometheus.MustRegister(processingTime)
59  }
60
61  func (c *CollectorServer) dataHandler(w http.ResponseWriter, r *http.
      Request) {
62    timer := prometheus.NewTimer(processingTime)
63    defer timer.ObserveDuration()
64
65    opsProcessed.Inc()
66    opsCurrent.Inc()
67    defer opsCurrent.Dec()
68
69    if r.Method != http.MethodPost {
70      http.Error(w, WrongMethodErrorMsg, http.StatusMethodNotAllowed)
```

```
71      return
72    }
73
74    errorMsg, statusCode, body := getBody(r)
75    if errorMsg != "" {
76      log.Error("There was an error getting the body,", errorMsg)
77      http.Error(w, errorMsg, statusCode)
78      return
79    }
80
81    errorMsg, statusCode, id, payload := parseBody(body)
82    if errorMsg != "" {
83      log.Error("There was an error parsing the body,", errorMsg)
84      http.Error(w, errorMsg, statusCode)
85      return
86    }
87
88    err := c.Pub.Publish(id, payload)
89    if err != nil {
90      log.Error("There was an error publishing the message,", err)
91      http.Error(w, CannotPublishMessage, http.StatusInternalServerError)
92      return
93    }
94    w.WriteHeader(http.StatusAccepted)
95  }
96
97  func getBody(r *http.Request) (errorMsg string, statusCode int, body []
      byte) {
98    body, err := ioutil.ReadAll(r.Body)
99    if err != nil {
100     return EmptyBodyMsg, http.StatusUnprocessableEntity, nil
101   }
102   if string(body) == "" {
103     return EmptyBodyMsg, http.StatusUnprocessableEntity, nil
104   }
105   return "", 0, body
106 }
107
108 func parseBody(body []byte) (errorMsg string, statusCode int, id string,
```

```
    payload json.RawMessage) {
109   var dataJSON DataJSON
110   err := json.Unmarshal(body, &dataJSON)
111   if err != nil {
112     return CannotDecodeErrorMsg, http.StatusBadRequest, "", nil
113   }
114
115   rawId := dataJSON["id"]
116   if len(rawId) == 0 {
117     return CannotFindIdErrorMsg, http.StatusBadRequest, "", nil
118   }
119   id = string(rawId)
120
121   payload = dataJSON["data"]
122   if len(payload) == 0 {
123     return CannotFindDataErrorMsg, http.StatusBadRequest, "", nil
124   }
125   return "", 0, id, payload
126 }
```

Listing C.3: Fichero `collector.go` en el que se definen las funciones del servidor HTTP.

```
package server

import (
  "bytes"
  "errors"
  "net/http"
  "net/http/httptest"
  "strings"
  "testing"

  "gitlab.com/iok8s/collector/internal/integration"
)

var tests = []struct {
  name              string
  jsonStr           []byte
  expectedStatus    int
  expectedErrorMsg  error
  publishError      error
}{
  {
    name:              "returns 202 if can parse the data field",
    jsonStr:           []byte(`{"data":{"data2": "too cold."}, "
   another_field": "too hot.", "id": "ok"}`),
    expectedStatus:    http.StatusAccepted,
    expectedErrorMsg: nil,
    publishError:      nil,
  },
  {
    name:              "returns 400 if cannot parse the data field",
    jsonStr:           []byte(`{"not_data":"too cold.", "id": "ok"}`),
    expectedStatus:    http.StatusBadRequest,
    expectedErrorMsg: errors.New(CannotFindDataErrorMsg),
    publishError:      nil,
  },
  {
    name:              "returns 400 if cannot parse the id field",
    jsonStr:           []byte(`{"not_data":"too cold", "not_id": "ok"}`),
    expectedStatus:    http.StatusBadRequest,
```

```
39      expectedErrorMsg: errors.New(CannotFindIdErrorMsg),
40      publishError:     nil,
41    },
42    {
43      name:             "returns 400 if cannot decode a json",
44      jsonStr:          []byte(`{"not data": `),
45      expectedStatus:   http.StatusBadRequest,
46      expectedErrorMsg: errors.New(CannotDecodeErrorMsg),
47      publishError:     nil,
48    },
49    {
50      name:             "returns 422 if body is empty",
51      jsonStr:          []byte(``),
52      expectedStatus:   http.StatusUnprocessableEntity,
53      expectedErrorMsg: errors.New(EmptyBodyMsg),
54      publishError:     nil,
55    },
56    {
57      name:             "returns 500 if cannot publish",
58      jsonStr:          []byte(`{"data": "ok", "id": "3"}`),
59      expectedStatus:   http.StatusInternalServerError,
60      expectedErrorMsg: errors.New(CannotPublishMessage),
61      publishError:     errors.New("error"),
62    },
63  }
64
65  func TestPOSTData(t *testing.T) {
66    for _, tt := range tests {
67      server := initServer(t, tt.publishError)
68      t.Run(tt.name, func(t *testing.T) {
69        request, err := integration.MakeRequest("/collect", tt.jsonStr)
70        if err != nil {
71          t.Errorf("got error while making the request, %v", err)
72        }
73        response := httptest.NewRecorder()
74
75        server.ServeHTTP(response, request)
76
77        assertStatus(t, response.Code, tt.expectedStatus)
```

```
78        assertErrorMessage(t, response.Body.String(), tt.expectedErrorMsg)
79      })
80    }
81  }
82
83  func TestGETData(t *testing.T) {
84    t.Run("cannot use other method than post i.e get", func(t *testing.T)
       {
85      jsonStr := []byte(`{"data": "value"}`)
86      request, _ := http.NewRequest(http.MethodGet, "/collect", bytes.
       NewBuffer(jsonStr))
87      request.Header.Set("Content-Type", "application/json")
88      response := httptest.NewRecorder()
89
90      server := initServer(t, nil)
91      server.ServeHTTP(response, request)
92
93      assertStatus(t, response.Code, http.StatusMethodNotAllowed)
94      assertErrorMessage(t, response.Body.String(), errors.New(
       WrongMethodErrorMsg))
95    })
96  }
97
98  func assertStatus(t *testing.T, got, want int) {
99    if got != want {
100     t.Errorf("got status %v, want %v", got, want)
101   }
102 }
103
104 func assertErrorMessage(t *testing.T, a string, b error) {
105   a = strings.TrimSuffix(a, "\n") // http.Error adds a "\n" at the end
106   if b == nil {                   // if the error is nil we cannot use
       error.Error()
107     if a != "" {
108       t.Errorf("got error message '%v', want ''", a)
109     }
110     return
111   }
112
```

```
113   if strings.Compare(a, b.Error()) != 0 {
114     t.Errorf("got error message '%v', want '%v'", a, b)
115   }
116 }
117
118 func initServer(t *testing.T, expectedErr error) *CollectorServer {
119   s := integration.StubPublisher{
120     Err: expectedErr,
121   }
122   server := &CollectorServer{&s}
123   return server
124 }
```

Listing C.4: Fichero `collector_test.go` en el que se someten a test los diferentes componentes de `collector.go`.

C.3. Publicador de Kafka

```
1  package server
2
3  import (
4    "context"
5    "encoding/json"
6    "strings"
7    "time"
8
9    log "github.com/sirupsen/logrus"
10
11   kafka "github.com/segmentio/kafka-go"
12 )
13
14 type KafkaPublisher struct {
15   conn   *kafka.Conn
16   config ConnectionConfig
17 }
18
19 type ConnectionConfig struct {
20   address   string
```

```
  topic     string
  deadline  time.Duration
  partition int
}

func NewKafkaPublisher(address, topic string, deadline time.Duration,
    partition int) (Publisher, error) {
  config := ConnectionConfig{
    address:   address,
    topic:     topic,
    deadline:  deadline,
    partition: partition,
  }

  conn, err := GetConnection(config)
  if err != nil {
    log.Error("failed to create a connector: ", err)
    return nil, err
  }
  return &KafkaPublisher{conn, config}, nil
}

func GetConnection(config ConnectionConfig) (*kafka.Conn, error) {
  conn, err := kafka.DialLeader(context.Background(), "tcp", config.
    address, config.topic, config.partition)
  if err != nil {
    log.Error("failed to dial kafka leader, check the address of the
    cluster: ", err)
    return nil, err
  }
  err = conn.SetWriteDeadline(time.Now().Add(config.deadline))
  if err != nil {
    log.Error("failed to set write deadline: ", err)
    return nil, err
  }
  return conn, nil
}

func (p *KafkaPublisher) Publish(id string, payload interface{}) error {
```

```
57    message, err := p.encodePayload(id, payload)
58    if err != nil {
59      log.Error("failed to encode message: ", err)
60      return err
61    }
62    _, err = p.conn.WriteMessages(message)
63    if err != nil {
64      if strings.Contains(err.Error(), "i/o timeout") {
65        log.Warn("The connection has been timed out")
66        err := p.RenewConnection()
67        if err != nil {
68          log.Error("Error renewing the connection: ", err)
69          return err
70        }
71      } else if strings.Contains(err.Error(), "use of closed network
      connection") {
72        log.Warn("The connection is still closed, waiting before rewriting
      ")
73        time.Sleep(10 * time.Millisecond)
74      } else {
75        log.Error("Couldn't write the message, ", err)
76        return err
77      }
78
79      _, err = p.conn.WriteMessages(message)
80      if err != nil {
81        log.Error("Couldn't write the message at second attemp: ", err)
82        return err
83      }
84    }
85
86    log.Debug("Message written!")
87    return nil
88  }
89
90  func (p *KafkaPublisher) RenewConnection() error {
91    log.Info("Renewing")
92    p.conn.Close()
93    new_conn, err := GetConnection(p.config)
```

```
94    p.conn = new_conn
95    return err
96  }
97
98  func (p *KafkaPublisher) Destroy() error {
99    log.Debug("Destroying the publisher")
100   return p.conn.Close()
101 }
102
103 func (p *KafkaPublisher) encodePayload(id string, payload interface{}) (
        kafka.Message, error) {
104   log.Debug("Encoding the payload")
105   m, err := json.Marshal(payload)
106   if err != nil {
107     return kafka.Message{}, err
108   }
109
110   return kafka.Message{
111     Key:   []byte(id),
112     Value: m,
113   }, nil
114 }
```

Listing C.5: Fichero `kafka-publisher.go` en el que se define el componente que se va a utilizar para enviar mensajes al clúster de Kafka.

```go
package server

import (
  "fmt"
  "os"
  "testing"
  "time"
)

const topic = "topic_test"
const partition = 0

var address = fmt.Sprintf("%s:9092", getenv("KAFKA_HOST", "localhost"))

const deadline = 10 * time.Second

func TestKafkaPublisher(t *testing.T) {

  p, err := NewKafkaPublisher(address, topic, deadline, partition)

  if err != nil {
    t.Errorf("Cannot create a kafka publisher, %v", err)
  }

  id := "id_test"
  b := []byte("Hello World!")
  err = p.Publish(id, b)

  if err != nil {
    t.Error(err)
  }
}

func getenv(key, fallback string) string {
  value := os.Getenv(key)
  if len(value) == 0 {
    return fallback
  }
  return value
```

```
}
```

Listing C.6: Fichero `kafka-publisher_test.go` en el que se someten a test los diferentes componentes de `kafka-publisher.go`.

C.4. Benchmark del servidor

```
package integration

import (
  "bytes"
  "net/http"
)

func MakeRequest(url string, payload []byte) (*http.Request, error) {
  request, err := http.NewRequest(http.MethodPost, url, bytes.NewBuffer(
   payload))
  if err != nil {
    return nil, err
  }
  request.Header.Set("Content-Type", "application/json")
  return request, nil
}

type StubPublisher struct {
  Err error
}

func (s *StubPublisher) Publish(id string, payload interface{}) error {
  return s.Err
}

func (s *StubPublisher) Destroy() error {
  return nil
}
```

Listing C.7: Fichero `server-integration.go` en el que se define una función de ayuda para enviar peticiones y el `StubPublisher` que se utiliza en `collector_test.go`.

```
package integration

import (
  "net/http"
  "net/http/httptest"
  "testing"

  "gitlab.com/iok8s/collector/internal/emulator"
  "gitlab.com/iok8s/collector/internal/server"
)

func BenchmarkMakeRequest(b *testing.B) {
  d := emulator.NewGeneralDevice()
  publisher := StubPublisher{
    Err: nil,
  }
  server := &server.CollectorServer{Pub: &publisher}

  for i := 0; i < b.N; i++ {
    m := d.Measure()
    p, err := d.GeneratePayload(m)
    if err != nil {
      b.Errorf("error generating payload, %v", err)
    }
    req, err := MakeRequest("/collect", p)
    if err != nil {
      b.Errorf("error generating the request %v", err)
    }
    response := httptest.NewRecorder()

    server.ServeHTTP(response, req)
    if response.Code != http.StatusAccepted {
      b.Errorf("got status %v, want %v", response.Code, http.
   StatusAccepted)
    }
  }
```

```
}
```

Listing C.8: Fichero `server-integration_test.go` en el que se realiza un *benchmark* de `collector.go`.

C.5. Ejecutables

```
package main

import (
  "flag"
  "net/http"
  "os"
  "time"

  log "github.com/sirupsen/logrus"

  "gitlab.com/iok8s/collector/internal/server"
)

var certFile = getenv("CERT_FILE", "cmd/collector/localhost.crt")
var keyFile = getenv("KEY_FILE", "cmd/collector/localhost.key")

const defaultTopic = "topic_test"
const defaultPartition = 0
const defaultAddress = "localhost:9092"
const defaultDeadline = 5
const defaultServingAddress = ":5000"

var topic string
var partition int
var address string
var intDeadline int
var servingAddress string

func main() {
  flag.StringVar(&topic, "topic", defaultTopic, "Kafka topic where
   posted data will be sent")
```

```
31    flag.IntVar(&partition, "partition", defaultPartition, "Kafka topic
       partition where messages will be sent")
32    flag.StringVar(&address, "kafkaAddress", defaultAddress, "Kafka
       cluster address")
33    flag.IntVar(&intDeadline, "deadline", defaultDeadline, "deadline for
       publishing a message to kafka in seconds")
34    flag.StringVar(&servingAddress, "serverAddress", defaultServingAddress
       , "address where the server will listen")
35
36    deadline := time.Duration(intDeadline) * time.Second
37
38    flag.Parse()
39
40    p, err := server.NewKafkaPublisher(address, topic, deadline, partition
       )
41    if err != nil {
42      log.Fatal("cannot initiate the kafka publisher")
43    }
44
45    defer p.Destroy() // Close the connection when leaving main
46    server := &server.CollectorServer{Pub: p}
47
48    log.Infof("Serving at %q", servingAddress)
49    log.Fatal(http.ListenAndServeTLS(servingAddress, certFile, keyFile,
       server))
50  }
51
52  func getenv(key, fallback string) string {
53    value := os.Getenv(key)
54    if len(value) == 0 {
55      return fallback
56    }
57    return value
58  }
```

Listing C.9: Fichero `main.go` para el módulo `collector` en el que se compila un ejecutable que despliega un servidor HTTP con la lógica completa de la API.

```go
package main

import (
  "crypto/tls"
  "flag"
  "fmt"
  "io/ioutil"
  "log"
  "math/rand"
  "net/http"
  "sync"
  "sync/atomic"
  "time"

  "gitlab.com/iok8s/collector/internal/emulator"
  "gitlab.com/iok8s/collector/internal/integration"
)

const defaultNDevices = 1
const defaultTimeout = 10
const defaultMaxTimeBetweenMessages = 1
const defaultPostDataUrl = "https://localhost:5000/collect"

var nDevices int
var timeout int
var maxTimeBetweenMessages int
var dStack []emulator.Device
var postDataUrl string

func main() {
  flag.IntVar(&nDevices, "nDevices", defaultNDevices, "Number of devices
    to emulate")
  flag.IntVar(&timeout, "timeout", defaultTimeout, "Default timeout of
   the program")
  flag.IntVar(&maxTimeBetweenMessages, "maxTimeMsg",
   defaultMaxTimeBetweenMessages, "Maximum miliseconds between
   consecutive messages from the same device")
  flag.StringVar(&postDataUrl, "url", defaultPostDataUrl, "Collector URL
   ")
```

```
35    flag.Parse()
36
37    loop_timeout := time.Duration(timeout) * time.Second
38
39    log.Printf("Emulating %d devices", nDevices)
40
41    for i := 1; i <= nDevices; i++ {
42      dStack = append(dStack, emulator.NewGeneralDevice())
43    }
44
45    var wg sync.WaitGroup
46    wg.Add(nDevices)
47
48    var ops uint64
49
50    // Ignore self signed certificates
51    http.DefaultTransport.(*http.Transport).TLSClientConfig = &tls.Config{
       InsecureSkipVerify: true}
52
53    for _, d := range dStack {
54      go func(d emulator.Device) {
55        defer wg.Done()
56        for start := time.Now(); time.Since(start) < loop_timeout; {
57          emulate(d)
58          atomic.AddUint64(&ops, 1)
59          time.Sleep(time.Duration(rand.Float32()) * time.Millisecond)
60        }
61      }(d)
62    }
63    wg.Wait()
64
65    fmt.Println("Number of requests:", ops)
66  }
67
68  func emulate(d emulator.Device) {
69    client := http.Client{}
70    p, err := d.GeneratePayload(d.Measure())
71    if err != nil {
72      log.Fatalf("error generating Payload for device %v", d.Id)
```

```
  }

  req, err := integration.MakeRequest(postDataUrl, p)
  if err != nil {
    log.Fatalf("error generating the request, %v", err)
  }

  resp, err := client.Do(req)
  if err != nil {
    log.Fatalf("error sending request for device %v, %v", d.Id, err)
  }

  if resp == nil {
    log.Fatalf("couldn't get a response for device %v", d.Id)
  } else {
    if resp.StatusCode != http.StatusAccepted {
      defer resp.Body.Close()
      body, _ := ioutil.ReadAll(resp.Body)
      log.Printf("response is: %q", string(body))
      log.Fatalf("got status %v, want %v", resp.StatusCode, http.
    StatusAccepted)
    }
  }
}
```

Listing C.10: Fichero `main.go` para el módulo `emulator` en el que se compila un ejecutable que simula varios dispositivos que publican datos de varios tipos cada cierto tiempo.

```
FROM golang:alpine AS build

WORKDIR /go/src/gitlab.com/iok8s/collector
COPY . .

# Compile the code and save it to the GOPATH
RUN CGO_ENABLED=0 go build -o /go/bin/collector cmd/collector/main.go

# Generate the certificates
RUN apk upgrade --update-cache --available && \
    apk add openssl && \
    rm -rf /var/cache/apk/*
```

```
13
14 RUN openssl req  -new  -newkey rsa:2048 \
15    -nodes  -keyout cmd/collector/localhost.key \
16    -subj /CN=localhost/ \
17    -out cmd/collector/localhost.csr
18
19 RUN openssl  x509  -req  -days 365  -in cmd/collector/localhost.csr  -
      signkey cmd/collector/localhost.key  -out cmd/collector/localhost.crt
20
21 FROM scratch
22 COPY --from=build /go/bin/collector /go/bin/collector
23 COPY --from=build /go/src/gitlab.com/iok8s/collector/cmd/collector/
      localhost* /go/bin/
24 ENV CERT_FILE=/go/bin/localhost.crt
25 ENV KEY_FILE=/go/bin/localhost.key
26
27 ENTRYPOINT ["/go/bin/collector"]
```

Listing C.11: Fichero `Dockerfile` para crear la imagen de la API.

C.6. Integración continua

```
stages:
  - test

services:
  - docker:dind

stages:
- test

test:
  stage: test
  image: docker/compose:alpine-1.27.4
  script:
    - cd deployments
    - docker-compose -f docker-compose.yml build
    - docker-compose -f docker-compose.yml up --exit-code-from go
   -test --abort-on-container-exit
```

Listing C.12: Fichero `.gitlab-ci.yml` para la integración continua del código del repositorio.

```yaml
version: "3.8"

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:6.0.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    networks:
      - kafka-network
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
      ZOOKEEPER_LOG4J_ROOT_LOGLEVEL: WARN

  kafka:
    image: confluentinc/cp-kafka:6.0.0
    hostname: kafka
    container_name: kafka
    depends_on:
      - zookeeper
    networks:
      - kafka-network
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_NUM_PARTITIONS: 3
      KAFKA_HEAP_OPTS: -Xmx512M -Xms512M
      KAFKA_LOG4J_ROOT_LOGLEVEL: WARN
      KAFKA_LOG4J_LOGGERS: "org.apache.zookeeper=WARN,org.apache.
    kafka=WARN,kafka=WARN,kafka.cluster=WARN,kafka.controller=WARN,
    kafka.coordinator=WARN,kafka.log=WARN,kafka.server=WARN,kafka.
    zookeeper=WARN,state.change.logger=WARN"

```

```
37
38   go-test:
39     image: golang:1.16.3-buster
40     networks:
41       - kafka-network
42     depends_on:
43       - kafka
44       - zookeeper
45     volumes:
46       - ../:/go/app
47     environment:
48       KAFKA_HOST: kafka
49     working_dir: /go/app
50     command: bash -c "sleep 15s && go test ./... -bench=. "
51
52 networks:
53   kafka-network:
```

Listing C.13: Fichero `docker-compose.yml` que utiliza `.gitlab-ci.yml` para probar el código de la API desplegando a su vez un clúster de Kafka en Docker. De esta forma se puede probar también el publicador de Kafka en lugar de simularlo.

Apéndice D

Código del módulo de reglas

D.1. Motor de reglas

```
1  package main
2
3  import (
4    "context"
5    "fmt"
6    "strconv"
7    "strings"
8
9    "github.com/go-redis/redis/v8"
10   log "github.com/sirupsen/logrus"
11 )
12
13 type Rule struct {
14   Blocked bool     `json:"blocked"`
15   Fields  []string `json:"fields"`
16   MaxSize int      `json:"maxSize"`
17 }
18
19 type RuleEngine struct {
20   dbClient *redis.Client
21 }
22
23 func (e RuleEngine) fieldExist(id string, field string) (bool, error) {
```

```
24    exists, err := e.dbClient.HExists(context.Background(), fmt.Sprintf("
      rule:%s", id), field).Result()
25    switch {
26    case err == redis.Nil:
27      // id does not exist
28      return false, fmt.Errorf("id %q doesn't exist", id)
29    case err != nil:
30      return false, fmt.Errorf("error getting the field %q value from id %
      q, %v", field, id, err)
31    case !exists:
32      return false, nil
33    }
34    return true, nil
35  }
36
37  // If the id doesn't have a "blocked" field or doesn't exist, it is not
      blocked by default
38  func (e RuleEngine) idIsBlocked(id string) (bool, error) {
39    val, err := getFieldValue(e, id, "blocked")
40    if err != nil {
41      return false, err
42    }
43    if val == "" {
44      return false, nil
45    }
46
47    blocked, err := strconv.ParseBool(val)
48    switch {
49    case err != nil:
50      // Block the device as it has a blocked field which is not "false"
      or 0
51      return true, fmt.Errorf("error parsing the blocked field value %q, %
      q", val, err)
52    case blocked:
53      return true, nil
54    }
55    return false, nil
56  }
57
```

```
58 func (e RuleEngine) getMaxSize(id string) (int, error) {
59   val, err := getFieldValue(e, id, "maxSize")
60   if err != nil {
61     return 0, err
62   }
63   if val == "" {
64     return 0, err
65   }
66   maxSize, err := strconv.Atoi(val)
67   if err != nil {
68     return 0, fmt.Errorf("error parsing the maxSize field value %q, %q",
       val, err)
69   }
70   return maxSize, nil
71 }
72
73 // Returns true if all the fields in the database match the JSON fields
74 func (e RuleEngine) blockedByFields(id string, fields []string) (bool,
     error) {
75   val, err := getFieldValue(e, id, "fields")
76   if err != nil {
77     return false, err
78   }
79   if val == "" {
80     return false, nil
81   }
82   msgFields := strings.Split(val, ",")
83   if len(msgFields) < 1 {
84     return false, fmt.Errorf("invalid fields value %q", val)
85   }
86
87   for _, field := range fields {
88     if !contains(msgFields, field) {
89       // If the field value in the database isn't inside the fields
     slice, block the message
90       return true, nil
91     }
92   }
93   return false, nil
```

```
94 }
95
96 func (e RuleEngine) getRule(id string) (Rule, error) {
97   var rule Rule
98   val, err := e.dbClient.HGetAll(context.Background(), fmt.Sprintf("rule
     :%s", id)).Result()
99   if err == nil && len(val) == 0 {
100     log.Debugf("id %q doesn't exist", id)
101     return Rule{}, nil
102   }
103   if err != nil {
104     return Rule{}, fmt.Errorf("error looking for the rule for id %q, %q"
     , id, err)
105   }
106   rule.Blocked, err = strconv.ParseBool(val["blocked"])
107   if err != nil {
108     return Rule{}, fmt.Errorf("error parsing the blocked field %q for
     the rule for id %q, %q", val["blocked"], id, err)
109   }
110   rule.Fields = strings.Split(val["fields"], ",")
111   rule.MaxSize, err = strconv.Atoi(val["maxSize"])
112   if err != nil {
113     return Rule{}, fmt.Errorf("error parsing the maxSize field %q for
     the rule for id %q, %q", val["maxSize"], id, err)
114   }
115   return rule, nil
116 }
117
118 func (e RuleEngine) setRule(id string, rule Rule) error {
119   _, err := e.dbClient.HSet(context.Background(), fmt.Sprintf("rule:%s",
      id), "blocked", strconv.FormatBool(rule.Blocked), "fields", strings.
     Join(rule.Fields, ","), "maxSize", strconv.Itoa(rule.MaxSize)).Result
     ()
120   return err
121 }
122
123 func (e RuleEngine) listRules() (map[string]Rule, error) {
124   var cursor uint64
125   var keys []string
```

```
126   ctx := context.Background()
127   for {
128     var err error
129     var subKeys []string
130     subKeys, cursor, err = e.dbClient.Scan(ctx, cursor, "rule:*", 10).
      Result()
131     if err != nil {
132       panic(err)
133     }
134     keys = append(keys, subKeys...)
135     if cursor == 0 {
136       break
137     }
138   }
139   rules := make(map[string]Rule)
140   for _, key := range keys {
141     rule, err := e.getRule(strings.TrimPrefix(key, "rule:"))
142     if err != nil {
143       return map[string]Rule{}, err
144     }
145     rules[strings.TrimPrefix(key, "rule:")] = rule
146   }
147   return rules, nil
148 }
149
150 func getFieldValue(e RuleEngine, id string, field string) (string, error
     ) {
151   fieldExists, err := e.fieldExist(id, field)
152   if err != nil {
153     return "", fmt.Errorf("error looking for the %q field, %q", field,
      err)
154   }
155   if !fieldExists {
156     return "", nil
157   }
158
159   val, err := e.dbClient.HGet(context.Background(), fmt.Sprintf("rule:%s
      ", id), field).Result()
160   switch {
```

```
161   case err == redis.Nil:
162     // id does not exist
163     return "", fmt.Errorf("id %q doesn't exist", id)
164   case err != nil:
165     return "", fmt.Errorf("error getting id, %q", err)
166   }
167
168   return val, nil
169 }
170
171 func contains(s []string, e string) bool {
172   for _, a := range s {
173     if a == e {
174       return true
175     }
176   }
177   return false
178 }
```

Listing D.1: Fichero `engine.go` en el que se define el motor de reglas junto con las funciones encargadas de crear, listar y modificar las reglas almacenadas en la base de datos.

```
package main

import (
  "context"
  "fmt"
  "testing"
)

func TestFieldExist(t *testing.T) {
  type test struct {
    name     string
    database map[string]Rule
    field    string
    want     bool
  }
  testCases := []test{
    {
      name:     "id doesn't exist",
      database: nil,
      field:    "blocked",
      want:     false,
    },
    {
      name:     "id exists and field exists",
      database: map[string]Rule{id: nonblockedRule},
      field:    "blocked",
      want:     true,
    },
    {
      name:     "id exists and field doesn't exist",
      database: map[string]Rule{id: nonblockedRule},
      field:    "dummyblocked",
      want:     false,
    },
  }

  for _, testCase := range testCases {
    t.Run(testCase.name, func(t *testing.T) {
      rdb := initRedisMock(testCase.database)
```

```
      defer tearDownRedisMock()
      s := RuleEngine{rdb}

      found, err := s.fieldExist(id, testCase.field)
      checkResult(t, err, false, found, testCase.want)
    })
  }
}

func TestGetRule(t *testing.T) {
  type test struct {
    name     string
    database map[string]Rule
    want     Rule
  }
  testCases := []test{
    {
      name:     "id doesn't exist",
      database: nil,
      want:     Rule{},
    },
    {
      name:     "id exists",
      database: map[string]Rule{id: validFieldsRule},
      want:     validFieldsRule,
    },
  }

  for _, testCase := range testCases {
    t.Run(testCase.name, func(t *testing.T) {
      rdb := initRedisMock(testCase.database)
      defer tearDownRedisMock()
      s := RuleEngine{rdb}

      rule, err := s.getRule(id)
      checkResult(t, err, false, rule, testCase.want)
    })
  }
}
```

```
79
80 func TestIdIsBlocked(t *testing.T) {
81   type test struct {
82     name     string
83     database map[string]Rule
84     id       string
85     want     bool
86   }
87   testCases := []test{
88     {
89       name:     "id doesn't exist",
90       database: map[string]Rule{id: nonblockedRule},
91       id:       "dummyid",
92       want:     false,
93     },
94     {
95       name:     "id exists and isn't blocked",
96       database: map[string]Rule{id: nonblockedRule},
97       id:       id,
98       want:     false,
99     },
100    {
101      name:     "id exists and is blocked",
102      database: map[string]Rule{id: blockedRule},
103      id:       id,
104      want:     true,
105    },
106  }
107  for _, testCase := range testCases {
108    t.Run(testCase.name, func(t *testing.T) {
109      rdb := initRedisMock(testCase.database)
110      defer tearDownRedisMock()
111      s := RuleEngine{rdb}
112
113      blocked, err := s.idIsBlocked(testCase.id)
114      checkResult(t, err, false, blocked, testCase.want)
115    })
116  }
117  t.Run("if blocked field doesn't exist the id is not blocked", func(t *
```

```
    testing.T) {
118     rdb := initRedisMock(map[string]Rule{"": nonblockedRule})
119     defer tearDownRedisMock()
120     rdb.HSet(context.Background(), fmt.Sprintf("rule:%s", id), "
        dummyblocked", "true")
121
122     s := RuleEngine{rdb}
123
124     blocked, err := s.idIsBlocked(id)
125     checkResult(t, err, false, blocked, false)
126   })
127 }
128
129 func TestGetMaxSize(t *testing.T) {
130   type test struct {
131     name          string
132     database      map[string]Rule
133     id            string
134     size          int
135     expectedError bool
136     want          int
137   }
138   testCases := []test{
139     {
140       name:          "id doesn't exist",
141       database:      map[string]Rule{id: validSizeRule},
142       id:            "dummyid",
143       size:          0,
144       expectedError: false,
145       want:          0,
146     },
147     {
148       name:          "id exists and size is lesser than maxSize",
149       database:      map[string]Rule{id: validSizeRule},
150       id:            id,
151       expectedError: false,
152       want:          200,
153     },
154     {
```

```
155       name:           "id exists and size is higher than maxSize",
156       database:       map[string]Rule{id: validSizeRule},
157       id:             id,
158       expectedError: false,
159       want:           200,
160     },
161   }
162   for _, testCase := range testCases {
163     t.Run(testCase.name, func(t *testing.T) {
164       rdb := initRedisMock(testCase.database)
165       defer tearDownRedisMock()
166       s := RuleEngine{rdb}
167
168       maxSize, err := s.getMaxSize(testCase.id)
169       checkResult(t, err, testCase.expectedError, maxSize, testCase.want
    )
170     })
171   }
172 }
173
174 func TestBlockedByFields(t *testing.T) {
175   type test struct {
176     name          string
177     database      map[string]Rule
178     id            string
179     fields        []string
180     expectedError bool
181     want          bool
182   }
183   testCases := []test{
184     {
185       name:           "id doesn't exist",
186       database:       map[string]Rule{id: validFieldsRule},
187       id:             "dummyid",
188       fields:         []string{""},
189       expectedError: false,
190       want:           false,
191     },
192     {
```

```
193       name:           "id exists fields value is empty",
194       database:       map[string]Rule{id: emptyFieldsRule},
195       id:             id,
196       fields:         []string{""},
197       expectedError: false,
198       want:           false,
199     },
200     {
201       name:           "id exists and fields don't match",
202       database:       map[string]Rule{id: validFieldsRule},
203       id:             id,
204       fields:         []string{"unwanted1", "unwanted2"},
205       expectedError: false,
206       want:           true,
207     },
208     {
209       name:           "id exists and fields match",
210       database:       map[string]Rule{id: validFieldsRule},
211       id:             id,
212       fields:         []string{"wanted1", "wanted2"},
213       expectedError: false,
214       want:           false,
215     },
216   }
217   for _, testCase := range testCases {
218     t.Run(testCase.name, func(t *testing.T) {
219       rdb := initRedisMock(testCase.database)
220       defer tearDownRedisMock()
221       s := RuleEngine{rdb}
222
223       blocked, err := s.blockedByFields(testCase.id, testCase.fields)
224       checkResult(t, err, testCase.expectedError, blocked, testCase.want
    )
225     })
226   }
227 }
228
229 func TestSetRule(t *testing.T) {
230   type test struct {
```

```
231     name           string
232     database       map[string]Rule
233     id             string
234     rule           Rule
235     expectedError  bool
236     want           map[string]Rule
237   }
238   testCases := []test{
239     {
240       name:          "id doesn't exist",
241       database:      map[string]Rule{},
242       id:            id,
243       rule:          validFieldsRule,
244       expectedError: false,
245       want:          map[string]Rule{id: validFieldsRule},
246     },
247     {
248       name:          "id exists",
249       database:      map[string]Rule{id: blockedRule},
250       id:            id,
251       rule:          validFieldsRule,
252       expectedError: false,
253       want:          map[string]Rule{id: validFieldsRule},
254     },
255   }
256   for _, testCase := range testCases {
257     t.Run(testCase.name, func(t *testing.T) {
258       rdb := initRedisMock(testCase.database)
259       defer tearDownRedisMock()
260       s := RuleEngine{rdb}
261
262       err := s.setRule(testCase.id, testCase.rule)
263       checkResult(t, err, testCase.expectedError, nil, nil)
264     })
265   }
266 }
267
268 func TestListRules(t *testing.T) {
269   type test struct {
```

```
    name     string
    database map[string]Rule
  }
  testCases := []test{
    {
      name:     "no ids",
      database: map[string]Rule{},
    },
    {
      name:     "id exists",
      database: map[string]Rule{id: validFieldsRule, "dummyid":
   validSizeRule},
    },
  }

  for _, testCase := range testCases {
    t.Run(testCase.name, func(t *testing.T) {
      rdb := initRedisMock(testCase.database)
      defer tearDownRedisMock()
      s := RuleEngine{rdb}

      rules, err := s.listRules()
      checkResult(t, err, false, rules, testCase.database)
    })
  }
}
```

Listing D.2: Fichero `engine_test.go` donde se prueba el funcionamiento de `engine.go`.

D.2. Servidor web

```
package main

import (
  "encoding/json"
  "errors"
  "html/template"
  "io/ioutil"
```

```
  "net/http"
  "reflect"

  log "github.com/sirupsen/logrus"
)

func init() {
  log.SetLevel(log.DebugLevel)
}

type RuleServer struct {
  ruleEngine *RuleEngine
}

type Results struct {
  Total int              `json:"total"`
  Rules map[string]Rule `json:"rules"`
}

var tpl = template.Must(template.ParseFiles("assets/index.html"))

func (c *RuleServer) ServeHTTP(w http.ResponseWriter, r *http.Request) {
  router := http.NewServeMux()
  router.Handle("/auth/", http.HandlerFunc(c.isAble))
  router.Handle("/rule/", http.HandlerFunc(c.manageRules))
  router.Handle("/rules", http.HandlerFunc(c.getRules))
  router.Handle("/", http.HandlerFunc(c.indexHandler))

  fs := http.FileServer(http.Dir("./assets"))
  router.Handle("/assets/", http.StripPrefix("/assets", fs))

  router.ServeHTTP(w, r)
}

func (c *RuleServer) indexHandler(w http.ResponseWriter, r *http.Request
    ) {
  log.Debug("webapp rules", r.Method)
  tpl = template.Must(template.ParseFiles("assets/index.html"))

```

```
46    rules, err := c.ruleEngine.listRules()
47    if err != nil {
48      log.Debugf("error getting rules: %v", err)
49      tpl.Execute(w, nil)
50      return
51    }
52    results := Results{
53      Total: len(rules),
54      Rules: rules,
55    }
56    tpl.Execute(w, results)
57  }
58
59  func (c *RuleServer) isAble(w http.ResponseWriter, r *http.Request) {
60    log.Debug("isAble", r.Method)
61    if r.Method != http.MethodGet {
62      log.Debugf("method %q not allowed", r.Method)
63      http.Error(w, http.StatusText(http.StatusMethodNotAllowed), http.
      StatusMethodNotAllowed)
64      return
65    }
66    // get id and check if it's blocked
67    id := r.URL.Path[len("/auth/"):]
68    blocked, err := c.ruleEngine.idIsBlocked(id)
69    if err != nil {
70      log.Debugf("error checking if id %q is blocked: %v", id, err)
71      http.Error(w, http.StatusText(http.StatusInternalServerError), http.
      StatusInternalServerError)
72      return
73    }
74    if blocked {
75      log.Debugf("id %q is blocked", id)
76      w.WriteHeader(http.StatusForbidden)
77      return
78    }
79
80    // check if request body is not too large
81    maxBytesSize, err := c.ruleEngine.getMaxSize(id)
82    if err != nil {
```

```
83    log.Debugf("error getting id %q maxSize: %v", id, err)
84    http.Error(w, http.StatusText(http.StatusInternalServerError), http.
      StatusInternalServerError)
85    return
86  }
87
88  if maxBytesSize != 0 {
89    r.Body = http.MaxBytesReader(w, r.Body, int64(maxBytesSize))
90  }
91  data, err := ioutil.ReadAll(r.Body)
92  if err != nil {
93    if err.Error() == "http: request body too large" {
94      w.WriteHeader(http.StatusForbidden)
95      log.Debug("request too big")
96      return
97    }
98    log.Debug("error reading body, ", err)
99    http.Error(w, http.StatusText(http.StatusInternalServerError), http.
      StatusInternalServerError)
100   return
101 }
102
103 if string(data) == "" {
104   log.Debug("empty body")
105   http.Error(w, "empty body", http.StatusBadRequest)
106   return
107 }
108
109 // check if it contains the necessary fields
110 var dataAsJson map[string]json.RawMessage
111 err = json.Unmarshal(data, &dataAsJson)
112 if err != nil {
113   log.Debug("cannot decode body ", err)
114   http.Error(w, "cannot decode body", http.StatusBadRequest)
115   return
116 }
117
118 var msgFields []string
119 for field := range dataAsJson {
```

```
120     msgFields = append(msgFields, field)
121   }
122
123   blocked, err = c.ruleEngine.blockedByFields(id, msgFields)
124   if err != nil {
125     log.Debugf("error checking if id %q fields %q areenough: %v", id,
      msgFields, err)
126     http.Error(w, http.StatusText(http.StatusInternalServerError), http.
      StatusInternalServerError)
127     return
128   }
129   if blocked {
130     log.Debugf("id %q fields %q are blocked", id, msgFields)
131     w.WriteHeader(http.StatusForbidden)
132     return
133   }
134
135   // if it has passed every check return an OK status
136   w.WriteHeader(http.StatusOK)
137 }
138
139 func (c *RuleServer) manageRules(w http.ResponseWriter, r *http.Request)
      {
140   log.Debug("manageRules", r.Method)
141   if r.Method == http.MethodGet {
142     c.getRule(w, r)
143     return
144   }
145   if r.Method == http.MethodPost {
146     c.setRule(w, r)
147     return
148   }
149   log.Debugf("method %q not allowed", r.Method)
150   http.Error(w, http.StatusText(http.StatusMethodNotAllowed), http.
      StatusMethodNotAllowed)
151 }
152
153 func (c *RuleServer) getRule(w http.ResponseWriter, r *http.Request) {
154   log.Debug("getRule", r.Method)
```

```
155   id := r.URL.Path[len("/rule/"):]
156   rule, err := c.ruleEngine.getRule(id)
157   if err != nil {
158     log.Debugf("error getting id %q rule: %v", id, err)
159     http.Error(w, http.StatusText(http.StatusInternalServerError), http.
      StatusInternalServerError)
160     return
161   }
162   if reflect.DeepEqual(rule, Rule{}) {
163     log.Debugf("id %q not found", id)
164     http.Error(w, http.StatusText(http.StatusNotFound), http.
      StatusNotFound)
165     return
166   }
167
168   json.NewEncoder(w).Encode(rule)
169   w.Header().Set("Content-Type", "application/json")
170 }
171
172 func (c *RuleServer) getRules(w http.ResponseWriter, r *http.Request) {
173   log.Debug("getRules", r.Method)
174   rules, err := c.ruleEngine.listRules()
175   if err != nil {
176     log.Debugf("error getting rules: %v", err)
177     http.Error(w, http.StatusText(http.StatusInternalServerError), http.
      StatusInternalServerError)
178     return
179   }
180
181   json.NewEncoder(w).Encode(rules)
182   w.Header().Set("Content-Type", "application/json")
183 }
184
185 func (c *RuleServer) setRule(w http.ResponseWriter, r *http.Request) {
186   log.Debug("setRule ", r.Method)
187   log.Debug("setting rule")
188
189   id := r.URL.Path[len("/rule/"):]
190
```

```
191   body, err := parseBody(r)
192   if err != nil {
193     log.Debug(err)
194     http.Error(w, err.Error(), http.StatusUnprocessableEntity)
195     return
196   }
197
198   var rule Rule
199   err = json.Unmarshal(body, &rule)
200   log.Debug(string(body))
201
202   if err != nil {
203     log.Debug("setRule ", "cannot unmarshall body ", err)
204     http.Error(w, http.StatusText(http.StatusUnprocessableEntity), http.
      StatusUnprocessableEntity)
205     return
206   }
207
208   err = c.ruleEngine.setRule(id, rule)
209   if err != nil {
210     http.Error(w, http.StatusText(http.StatusInternalServerError), http.
      StatusInternalServerError)
211     return
212   }
213
214   log.Debugf("rule %v set", rule)
215   w.Write([]byte("{}"))
216 }
217
218 func parseBody(r *http.Request) ([]byte, error) {
219   body, err := ioutil.ReadAll(r.Body)
220   defer r.Body.Close()
221   if err != nil {
222     return nil, err
223   }
224   if string(body) == "" {
225     return nil, errors.New("empty body")
226   }
227   return body, nil
```

```
228 }
```

Listing D.3: Fichero `server.go` en el que se definen los *endpoints* para utilizar el motor de reglas mediante HTTP.

```
package main

import (
  "bytes"
  "encoding/json"
  "fmt"
  "net/http"
  "net/http/httptest"
  "reflect"
  "strings"
  "testing"
)

func TestPostBlocked(t *testing.T) {
  t.Run("post method is not allowed", func(t *testing.T) {
    rdb := initRedisMock(nil)
    e := RuleEngine{rdb}
    server := &RuleServer{&e}

    request, _ := http.NewRequest(http.MethodPost, fmt.Sprintf("/auth/%s
    ", id), bytes.NewBuffer([]byte(`{"a": "b"}`)))
    request.Header.Set("Content-Type", "application/json")
    response := httptest.NewRecorder()

    server.ServeHTTP(response, request)

    assertStatus(t, response.Code, http.StatusMethodNotAllowed)

  })
}

func TestGetBlockedId(t *testing.T) {
  var tests = []struct {
    name           string
    jsonStr        []byte
    expectedStatus int
    database       map[string]Rule
  }{
    {
```

```
39      name:           "returns 403 if id is blocked",
40      jsonStr:        []byte(`{"a":"aa", "b": "bb"}`),
41      database:       map[string]Rule{id: blockedRule},
42      expectedStatus: http.StatusForbidden,
43    },
44    {
45      name:           "returns 403 if size is too big",
46      jsonStr:        make([]byte, 1000),
47      database:       map[string]Rule{id: validSizeRule},
48      expectedStatus: http.StatusForbidden,
49    },
50    {
51      name:           "returns 403 if not enough fields",
52      jsonStr:        []byte(`{"unwanted1": "a", "wanted2": "b"}`),
53      database:       map[string]Rule{id: validFieldsRule},
54      expectedStatus: http.StatusForbidden,
55    },
56    {
57      name:           "returns 400 if cannot read the body",
58      jsonStr:        []byte(`{""}`),
59      database:       map[string]Rule{id: validFieldsRule},
60      expectedStatus: http.StatusBadRequest,
61    },
62    {
63      name:           "returns 200 if id does not exist",
64      jsonStr:        []byte(`{"wanted1": "a", "wanted2": "b"}`),
65      database:       map[string]Rule{"dummyid": validFieldsRule},
66      expectedStatus: http.StatusOK,
67    },
68    {
69      name:           "returns 200 if id exists, is not blocked and
    message is valid",
70      jsonStr:        []byte(`{"wanted1": "a", "wanted2": "b"}`),
71      database:       map[string]Rule{"id": validFieldsRule},
72      expectedStatus: http.StatusOK,
73    },
74  }
75  for _, testCase := range tests {
76    t.Run(testCase.name, func(t *testing.T) {
```

```
77        rdb := initRedisMock(testCase.database)
78        e := RuleEngine{rdb}
79        server := &RuleServer{&e}
80
81        request, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("/auth/%
    s", id), bytes.NewBuffer(testCase.jsonStr))
82        request.Header.Set("Content-Type", "application/json")
83        response := httptest.NewRecorder()
84
85        server.ServeHTTP(response, request)
86
87        assertStatus(t, response.Code, testCase.expectedStatus)
88
89      })
90    }
91  }
92
93  func TestGetRuleId(t *testing.T) {
94    var tests = []struct {
95      name           string
96      expectedStatus int
97      database       map[string]Rule
98      want           string
99    }{
100     {
101       name:           "returns 200 and rule if exists",
102       database:       map[string]Rule{id: validFieldsRule},
103       expectedStatus: http.StatusOK,
104       want:           `{"blocked":false,"fields":["wanted1","wanted2"],"
    maxSize":200}`,
105     },
106     {
107       name:           "returns 400 if user don't exist",
108       database:       map[string]Rule{"dummyid": validFieldsRule},
109       expectedStatus: http.StatusNotFound,
110       want:           http.StatusText(http.StatusNotFound),
111     },
112   }
113   for _, testCase := range tests {
```

```
114     t.Run(testCase.name, func(t *testing.T) {
115       rdb := initRedisMock(testCase.database)
116       e := RuleEngine{rdb}
117       server := &RuleServer{&e}
118
119       request, _ := http.NewRequest(http.MethodGet, fmt.Sprintf("/rule/%
    s", id), nil)
120       request.Header.Set("Content-Type", "application/json")
121       response := httptest.NewRecorder()
122
123       server.ServeHTTP(response, request)
124
125       assertStatus(t, response.Code, testCase.expectedStatus)
126       if strings.TrimSuffix(response.Body.String(), "\n") != testCase.
    want {
127         t.Errorf("got %q want %q", response.Body, testCase.want)
128       }
129
130     })
131   }
132 }
133
134 func TestGetRules(t *testing.T) {
135   var tests = []struct {
136     name           string
137     expectedStatus int
138     database       map[string]Rule
139   }{
140     {
141       name:           "returns 200 and rules if exists",
142       database:       map[string]Rule{id: validFieldsRule, "dummyid":
    validSizeRule},
143       expectedStatus: http.StatusOK,
144     },
145     {
146       name:           "returns 200 and empty response if not exists",
147       database:       map[string]Rule{},
148       expectedStatus: http.StatusOK,
149     },
```

```
150   }
151   for _, testCase := range tests {
152     t.Run(testCase.name, func(t *testing.T) {
153       rdb := initRedisMock(testCase.database)
154       e := RuleEngine{rdb}
155       server := &RuleServer{&e}
156
157       request, _ := http.NewRequest(http.MethodGet, "/rules", nil)
158       request.Header.Set("Content-Type", "application/json")
159       response := httptest.NewRecorder()
160
161       server.ServeHTTP(response, request)
162
163       assertStatus(t, response.Code, testCase.expectedStatus)
164
165       var result map[string]Rule
166       err := json.NewDecoder(response.Body).Decode(&result)
167       if err != nil {
168         t.Error("error decoding json ", err)
169       }
170       if !reflect.DeepEqual(result, testCase.database) {
171         t.Errorf("got %v want %v", result, testCase.database)
172       }
173
174     })
175   }
176 }
177
178 func TestPostRuleId(t *testing.T) {
179   var tests = []struct {
180     name           string
181     expectedStatus int
182     database       map[string]Rule
183     jsonStr        []byte
184     want           Rule
185   }{
186     {
187       name:           "returns 200 if id doesn't exist and the rule is
    saved",
```

```
188         database:       map[string]Rule{},
189         jsonStr:        []byte(`{"blocked":false,"fields":["wanted1","
    wanted2"],"maxSize":200}`),
190         expectedStatus: http.StatusOK,
191         want:           validFieldsRule,
192       },
193       {
194         name:           "returns 200 if id exists and the rule is updated"
    ,
195         database:       map[string]Rule{id: emptyFieldsRule},
196         jsonStr:        []byte(`{"blocked":false,"fields":["wanted1","
    wanted2"],"maxSize":200}`),
197         expectedStatus: http.StatusOK,
198         want:           validFieldsRule,
199       },
200       {
201         name:           "returns 422 if rule is not valid",
202         database:       map[string]Rule{id: validFieldsRule},
203         jsonStr:        []byte(`{"blocked": 200}`),
204         expectedStatus: http.StatusUnprocessableEntity,
205         want:           validFieldsRule,
206       },
207     }
208     for _, testCase := range tests {
209       t.Run(testCase.name, func(t *testing.T) {
210         rdb := initRedisMock(testCase.database)
211         e := RuleEngine{rdb}
212         server := &RuleServer{&e}
213
214         request, _ := http.NewRequest(http.MethodPost, fmt.Sprintf("/rule
    /%s", id), bytes.NewBuffer(testCase.jsonStr))
215         request.Header.Set("Content-Type", "application/json")
216         response := httptest.NewRecorder()
217
218         server.ServeHTTP(response, request)
219
220         assertStatus(t, response.Code, testCase.expectedStatus)
221         savedRule, err := e.getRule(id)
222         if err != nil {
```

```
223           t.Errorf("got error %v while getting the rule", err)
224         }
225         if !reflect.DeepEqual(savedRule, testCase.want) {
226           t.Errorf("got %v want %v", savedRule, testCase.want)
227         }
228
229       })
230     }
231 }
232
233 func assertStatus(t *testing.T, got, want int) {
234   if got != want {
235     t.Errorf("got status %v, want %v", got, want)
236   }
237 }
```

Listing D.4: Fichero `server_test.go` en el que se definen llas pruebas de `server.go`

```go
package main

import (
  "context"
  "net/http"
  "os"

  log "github.com/sirupsen/logrus"

  "github.com/go-redis/redis/v8"
)

const servingAddress = ":3000"

func main() {
  var (
    host     = getEnv("REDIS_HOST", "localhost")
    port     = string(getEnv("REDIS_PORT", "6379"))
    password = getEnv("REDIS_PASSWORD", "")
  )

  rdb := redis.NewClient(&redis.Options{
    Addr:     host + ":" + port,
    Password: password,
    DB:       0})

  _, err := rdb.Ping(context.Background()).Result()
  if err != nil {
    log.Fatal("cannot connect to redis: ", err)
  }

  e := RuleEngine{rdb}
  server := &RuleServer{&e}

  log.Infof("Serving at %q", servingAddress)
  err = http.ListenAndServe(servingAddress, server)
  if err != nil {
    log.Fatal(err)
  }
```

```
}

func getEnv(key, defaultValue string) string {
  value := os.Getenv(key)
  if value == "" {
    return defaultValue
  }
  return value
}
```

Listing D.5: Fichero `main.go` que se utiliza para crear el ejecutable que levanta el servidor web.

D.3. Web estática

```
<!DOCTYPE html>
<html lang="es">
    <head>
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <title>Rule Engine</title>
        <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-+0n0xVW2eSR5OomGNYDnhzAbDsOXxcvSN1TPprVMTNDbiYZCxYbOOl7+AMvyTG2x" crossorigin="anonymous">

        <script src="https://code.jquery.com/jquery-3.5.1.min.js" integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0=" crossorigin="anonymous"></script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery.form/4.3.0/jquery.form.min.js" integrity="sha384-qlmct0AOBiA2VPZkMY3+2WqkHtIQ9lSdAsAn5RUJD/3vA5MKDgSGcdmIv4ycVxyn" crossorigin="anonymous"></script>
    </head>
    <body>
        <section id="cover">
            <div id="container-sm" class="container">
                <h1>Crear o editar una regla</h1>
                <form id="set_rule">
                    <div class="form-group">
                        <label for="device_id">Identificador del dispositivo</label>
                        <input
                            id="device_id"
                            type="search"
                            name="device_id"
                            placeholder="Identificador del dispositivo"
                            class="form-control"
                        />
                    </div>
```

```
27                  <div class="form-group">
28                      <label for="blocked">Dispositivo bloqueado</
    label>
29                      <select
30                          name="blocked"
31                          id="blocked"
32                          default="false"
33                          class="form-control">
34                          <option value="true">true</options>
35                          <option value="false">false</options>
36                      </select>
37                  </div>
38                  <div class="form-group">
39                      <label for="fields">Campos necesarios</label>
40                      <input
41                          id="fields"
42                          type="search"
43                          name="fields"
44                          placeholder="Campos necesarios"
45                          aria-describedby="fieldsHelp"
46                          class="form-control"
47                      />
48                      <small id="fieldsHelp" class="form-text text-
    muted">Separar con comas</small>
49                  </div>
50                  <div class="form-group">
51                      <label for="maxSize">Tamano maximo del mensaje (
    en bytes)</label>
52                      <input
53                          id="maxSize"
54                          type="number"
55                          name="maxSize"
56                          placeholder="Tamano maximo del mensaje"
57                          aria-describedby="maxSizeHelp"
58                          class="form-control"
59                      />
60                      <small id="maxSizeHelp" class="form-text text-
    muted">0 no limita el tamano</small>
61                  </div>
```

```
62
63                  <button type="submit" class="btn btn-primary">
    Guardar</button>
64              </form>
65          </div>
66      </section>
67      <section id="reglas">
68          <div id="container" class="container">
69              <h1>Todas las reglas</h1>
70              <p>En total hay {{ .Total }} reglas</p>
71              <table class="table table-hover">
72                  <thead>
73                      <tr>
74                          <th scope="col">Identificador del
    dispositivo</th>
75                          <th scope="col">Bloqueado</th>
76                          <th scope="col">Campos necesarios</th>
77                          <th scope="col">Tamano maximo del mensaje</
    th>
78                      </tr>
79                  </thead>
80                  <tbody>
81                      {{ range $key, $rule := .Rules }}
82                      <tr>
83                          <th scope="row">{{ $key }}</th>
84                          <td>{{ $rule.Blocked }}</td>
85                          <td>
86                              {{ range $field := $rule.Fields }}
87                              <p>{{ $field }}</p>
88                              {{end}}
89                          </td>
90                          <td>{{ $rule.MaxSize }}</td>
91                      </tr>
92                      {{end}}
93                  </tbody>
94              </table>
95          </div>
96      </section>
97   </body>
```

```
98
99      <script src="assets/form_controler.js" defer></script>
100
101 </html>
```

Listing D.6: Fichero `index.html` donde se define el contenido de la web estática utilizando las plantillas de Go

```
$('#set_rule').submit(function (e) {
    e.preventDefault();
    var theForm = { };
    $.each($('#set_rule').serializeArray(), function() {
        theForm[this.name] = this.value;
    });
    let device_id = theForm["device_id"]
    if (device_id == "") {
        alert("El identificador no puede estar vacio")
        return false
    }
    delete theForm['device_id'];
    theForm["blocked"] = (theForm["blocked"] == "true")
    theForm["fields"] = theForm["fields"].split(",")
    theForm["maxSize"] = parseInt(theForm["maxSize"])

    $.ajax({
        url: "/rule/" + device_id,
        data: JSON.stringify(theForm),
        type: 'POST',
        dataType: 'json',
        contentType: 'application/json',
        success: function (data, textStatus) {
            alert("Regla guardada")
            location.reload();
        },
        error: function (jqXHR, textStatus, errorThrown) {
            console.log(jqXHR, textStatus, errorThrown);
            alert("Error guardando la regla:")
        }
    });
    return false
});
```

Listing D.7: Fichero `form_controler.js` donde se modifica el formulario para que envíe un JSON en la petición POST en lugar de codificarlo como `x-www-form-urlencoded`.

D.4. Otros archivos

```
package main

import (
  "fmt"
  "reflect"
  "strconv"
  "strings"
  "testing"

  "github.com/alicebob/miniredis/v2"
  "github.com/go-redis/redis/v8"
)

const id = "random_id"

var redisServer *miniredis.Miniredis

var (
  nonblockedRule = Rule{
    Blocked: false,
    Fields:  []string{""},
    MaxSize: 0,
  }
  blockedRule = Rule{
    Blocked: true,
    Fields:  []string{""},
    MaxSize: 0,
  }
  validSizeRule = Rule{
    Blocked: false,
    Fields:  []string{""},
    MaxSize: 200,
  }
  validFieldsRule = Rule{
    Blocked: false,
    Fields:  []string{"wanted1", "wanted2"},
```

```
    MaxSize: 200,
  }
  emptyFieldsRule = Rule{
    Blocked: false,
    Fields:  []string{""},
    MaxSize: 200,
  }
)

func checkResult(t *testing.T, err error, expectedAnError bool, got
    interface{}, want interface{}) {
  if err != nil && !expectedAnError {
    t.Errorf("didn't expect an error %q", err)
  }

  if !reflect.DeepEqual(got, want) {
    t.Errorf("got %v want %v", got, want)
  }
}

func initRedisMock(devices map[string]Rule) *redis.Client {
  s, err := miniredis.Run()
  redisServer = s
  if err != nil {
    panic(err)
  }

  // Initialize the database
  for id, rule := range devices {
    redisServer.HSet(fmt.Sprintf("rule:%s", id), "blocked", strconv.
    FormatBool(rule.Blocked), "fields", strings.Join(rule.Fields, ","), "
    maxSize", strconv.Itoa(rule.MaxSize))
  }

  rdb := redis.NewClient(&redis.Options{
    Addr: redisServer.Addr(),
  })
  return rdb
}
```

```
73
74 func tearDownRedisMock() {
75   redisServer.Close()
76 }
```

Listing D.8: Fichero `testing.go` con algunas variables de ayuda para los tests.

```
FROM golang:alpine
RUN mkdir /app
COPY . /app
WORKDIR /app
RUN go build -o main .
CMD ["/app/main"]
```

Listing D.9: Fichero `Dockerfile` para crear una imagen Docker con el módulo de reglas.

Apéndice E

Código de la evaluación de la plataforma

E.1. Visualización en tiempo real

```
from kafka import KafkaConsumer  # Consumer de Kafka
import json                      # Para cargar el mensaje ya que esta
    serializado en JSON

from matplotlib import pyplot as plt
import base64
import io
import matplotlib.image as mpimg

def base64_to_image(image):
    i = base64.b64decode(image.split(",")[1])
    i = io.BytesIO(i)
    i = mpimg.imread(i, format='JPG')
    return i

consumer = KafkaConsumer(
    'topic_test',
    bootstrap_servers=['192.168.49.2:30029'],
    auto_offset_reset='earliest',  # Al usar latest lee los mensajes que
    aun no se han consumido
```

```
    enable_auto_commit=True,        # para hacer commits periodicos de los offsets y no duplicar mensajes
    group_id='my-group-id',         # nombre del grupo de consumidores
    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)

for event in consumer:  # Para cada evento que llega al consumidor
    base46_img = event.value["security camera"]
    temperature = float(event.value["temperature"])
    humidity = int(event.value["humidity"])
    img = base64_to_image(base46_img)

    plt.imshow(img, interpolation='nearest')
    plt.title("{:.2f}C, {}%".format(temperature, humidity))
    plt.xlabel("%s:%d:%d" % (event.topic, event.partition, event.offset))
    plt.xticks([])
    plt.yticks([])
    plt.pause(0.05)
plt.show()
```

Listing E.1: Fichero `realtime_visualizer.py` utilizado para visualizar el dato entrante a la plataforma en tiempo real.

```
from kafka import KafkaConsumer  # Consumer de Kafka
import json                      # Para cargar el mensaje ya que esta serializado en JSON

from matplotlib import pyplot as plt
import base64
import io
import matplotlib.image as mpimg

def base64_to_image(image):
    i = base64.b64decode(image.split(",")[1])
    i = io.BytesIO(i)
    i = mpimg.imread(i, format='JPG')
    return i

consumer = KafkaConsumer(
```

```
    'topic_test',
    bootstrap_servers=['192.168.49.2:30029'],
    auto_offset_reset='earliest',  # Al usar latest lee los mensajes que aun no se han consumido
    enable_auto_commit=True,       # para hacer commits periodicos de los offsets y no duplicar mensajes
    group_id='my-group-id',        # nombre del grupo de consumidores
    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)


fig, axs = plt.subplots(2, 4)
axs_ravel = axs.ravel()

i = 0

for event in consumer:  # Para cada evento que llega al consumidor
    base46_img = event.value["security camera"]
    temperature = float(event.value["temperature"])
    humidity = int(event.value["humidity"])
    img = base64_to_image(base46_img)

    axs_ravel[i].imshow(img, interpolation='nearest')
    axs_ravel[i].title.set_text("{:.2f}C, {}%".format(temperature, humidity))
    axs_ravel[i].set_xlabel("%s:%d:%d" % (event.topic, event.partition, event.offset))
    axs_ravel[i].axes.xaxis.set_ticklabels([])
    axs_ravel[i].axes.yaxis.set_ticklabels([])

    i += 1
    if i == 8:
        plt.show()
        quit()
```

Listing E.2: Fichero `frame_visualizer.py` que muestra los primeros 9 mensajes sin leer disponibles en el broker de Kafka.