UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Máster en Big Data y Data Science: ciencia e ingeniería de datos

TRABAJO FIN DE MÁSTER

Contenedores para el despliegue de funciones Lambda

Alejandro Gil Hernán
Tutor: Miguel Ángel Mora Rincón

Septiembre 2019

Contenedores para el despliegue de funciones Lambda

AUTOR: Alejandro Gil Hernán
TUTOR: Miguel Ángel Mora Rincón

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Septiembre de 2019

Resumen

Desde la llegada de Internet a nuestras vidas, la forma en que entendemos e interactuamos con los dispositivos electrónicos ha cambiado radicalmente. Hoy en día se conectan a internet de forma muy común dispositivos de electrónica de consumo como teléfonos móviles, impresoras, televisiones o cámaras IP entre otros.

Desde hace unos años, se empezó a hablar del IoT (Internet de las cosas), no siendo más que una apuesta a largo plazo, la cual se ha convertido en una absoluta realidad a día de hoy. Cada vez es más frecuente encontrarse con nuevos dispositivos capaces de conectarse a Internet, lo que posibilita la creación de sistemas mucho más sofisticados y complejos, permitiendo al usuario controlar, monitorizar e incluso interconectar de forma remota estos dispositivos desde cualquier parte del mundo.

Se podría definir el internet de las cosas como una gran red que aloja todo tipo de dispositivos, donde puede conectarse cualquier objeto como vehículos, electrodomésticos, sensores, o simplemente objetos tales como calzado o muebles. En definitiva, cualquier cosa que se pueda imaginar podría ser conectada a internet e interaccionar sin necesidad de la intervención humana. El objetivo por tanto es una interacción de máquina a máquina, o lo que se conoce como una interacción M2M (machine to machine).

Como la gran mayoría de avances tecnológicos conseguidos hasta la fecha, el IoT pretende facilitar la vida de las personas mejorando diferentes ámbitos de la vida cotidiana como seguridad, salud, comunicación o economía.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

Introducción

1.1 Motivación

Cada vez son más las soluciones y dispositivos IoT[1] que se introducen en nuestras vidas, probablemente nos encontremos en un punto de inflexión, pues disponemos de dispositivos hardware baratos, conexiones rápidas, software avanzado y plataformas de servicios en la nube que nos permiten tratar grandes cantidades de información de manera ágil, sencilla, potente y barata.

No obstante, el IoT es una de las mayores apuestas tecnológicas prevista para los próximos años. Se estima que habrá cerca de 50 mil millones de dispositivos conectados en 2020 repartidos en varios ámbitos como ciudades, industrias, agricultura, edificios, electricidad, vehículos, compra inteligente, medicina conectada, etc.

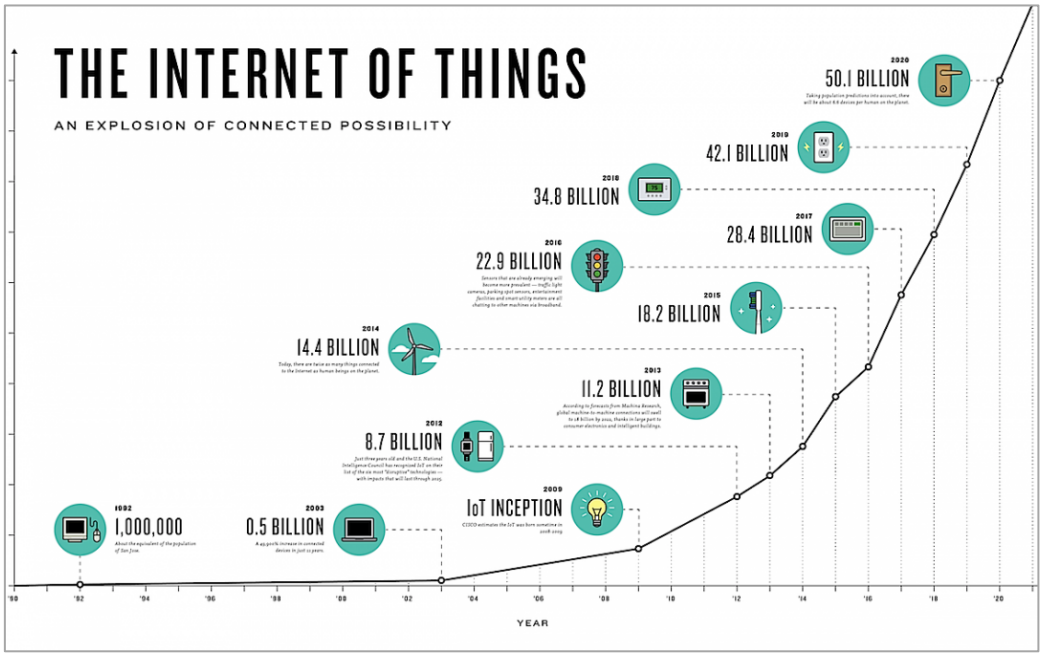

Figura 1. Previsión dispositivos IoT

Tanto empresas como usuarios finales podrán obtener grandes beneficios del IoT:

- **Automatización de procesos**: El hecho de que los dispositivos coexistan con los usuarios es una gran ventaja para liberarlos de realizar ciertas tareas, aumentando la eficiencia y rapidez
- **Obtención de información al instante**: Cuanta más información se posee, más fácil es tomar la decisión correcta, desde el ámbito cotidiano hasta las más grandes empresas (data driven)

[1] IoT challenges [20]
[1] What IoT needs to become a reality [15]

- **Facilidad de monitorización**: Gracias al envío de información en tiempo real, se consigue tener un seguimiento muy preciso de todos los parámetros de los dispositivos
- **Ahorro de costes**: Gracias a las ventajas mencionadas anteriormente, el Iot puede suponer un gran ahorro de costes.

Teniendo en cuenta estos factores, es necesario diseñar e implementar infraestructuras en las que desplegar aplicaciones IoT, teniendo en cuenta la escalabilidad y tolerancia a fallos necesaria para que estas puedas lidiar con gran cantidad de eventos.

1.2 Objetivos

Este trabajo se ha realizado con el objetivo principal de crear una plataforma partir de contenedores que sirva como base para desplegar aplicaciones IoT para ingestar, procesar y visualizar los eventos generados por los dispositivos. Dicha arquitectura está pensada para desplegarse en cualquier entorno, ya sea cloud u on-premise, evitando el bloqueo a un proveedor particular ya que las tecnologías usadas son agnósticas de la plataforma en la que son desplegadas.

Además, sobre la infraestructura se ha desarrollado una aplicación IoT basada en los eventos generados por vehículos para probar su funcionamiento.
Esta solución será desplegada en AWS, la plataforma de cloud pública de Amazon. El objetivo mencionado se lleva a cabo en diferentes fasesseparadoas por funcionalidad:

- Generación de eventos: Los eventos serán generados con un simulador desarrollado por AWS, no serán datos reales.
- Ingesta: Los eventos serán ingeridos y distribuidos a los consumidores por Kafka (herramienta de manipulación en tiempo real de mensajes).
- Procesamiento: Los consumidores procesan los eventos y los insertan en Elasticsearch (motor de búsqueda e indexación de texto).
- Visualización: Los diferentes eventos y métricas transferidas a Elasticsearch se visualizarán en Kibana (herramienta de visualización sobre el contenido indexado en Elasticsearch).

1.3 Organización de la memoria

El presente documento está dividido en cuatro partes. Primero, en el estado del arte, se hará una introducción a las tecnologías usadas en el trabajo como servicios cloud, contenedores o herramientas de procesamiento en tiempo real, a la vez que se presentan los distintos tipos de configuraciones que se van a implementar.

En el capítulo de diseño y desarrollo, se explica el proyecto realizado para la ingesta y procesamiento de eventos IoT en base a la teoría explicada, y mostrando las estructuras y componentes que se han implementado junto a los correspondientes diagramas.

La sección de resultados expone los experimentos llevados a cabo, incluyendo gráficos y dashboards representativos de los resultados obtenidos para las simulaciones realizadas.

Para finalizar, se muestran las conclusiones obtenidas comentando los objetivos a llevar a cabo en un futuro. En última instancia, los anexos proporcionan más información acerca de secciones el manual de instalación de componentes necesarios.

2 Estado del arte

2.1 Computación en la nube

El término computación en la nube hace referencia a la entrega de recursos TI virtualizados a través de internet. Consiste en entregar servicios informáticos bajo demanda con un modelo de pago por uso, como por ejemplo servicio de máquinas virtuales, contenedores o almacenamiento.
El modelo de computación en la nube está creciendo de manera muy notoria en los últimos años.

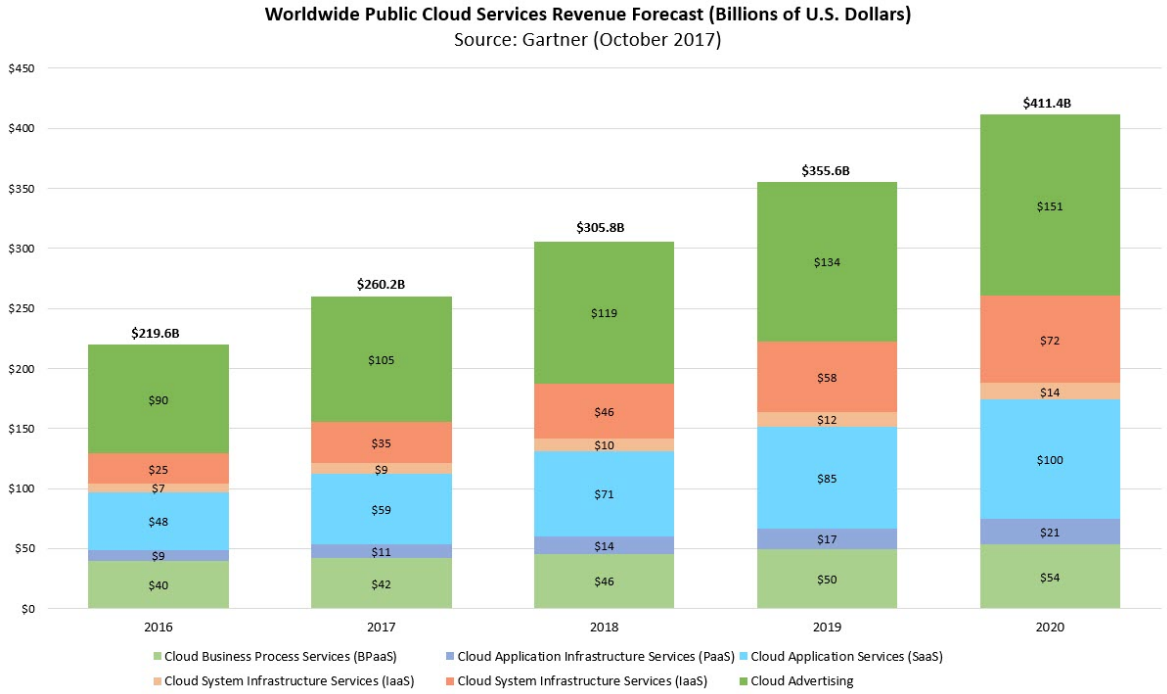

Figura 2. Evolución del beneficio asociado al cloud computing

Este modelo de computación a abierto un gran abanico de posibilidades. En lugar de tener que invertir en sus propios centros de datos, las empresas tienen la posibilidad de pagar únicamente por el tiempo que usen los servicios. La computación en la nube ofrece a pequeñas y grandes empresas multitud de beneficios como:

- Baja inversión inicial: una gran parte del presupuesto para TI se invierte en coste operacional en lugar de realizar un gran desembolso de capital inicial.

- Eficiencia de costes: todos los clientes se benefician de las economías de escalas que tienen los proveedores cloud.

- Escalabilidad y Elasticidad: los servicios cloud son altamente escalables, lo que permite hacer frente a cargas de trabajo realmente altas. Además, estos servicios son elásticos, lo que implica que ajustan su capacidad en base a la demanda que haya, lo que permite ahorrar costes y poder hacer frente a picos de carga inesperados.

- Alta disponibilidad y tolerancia a fallos: los proveedores cloud ofrecen la posibilidad de diseñar aplicaciones altamente disponibles, que no vean afectado su servicio incluso si una región completa se viene abajo.

- Facilidad de uso y mantenimiento: la gran diferencia con un centro de datos propio es que las empresas no tienen que mantener los servidores e instalaciones físicas. Además, una de las grandes virtudes que ofrecen los servicios gestionados es que es el propio proveedor cloud el que se encarga de actualizaciones de sistema operativo, parches, replicación de datos… etc.

- Capacidad de innovación: la flexibilidad propia de los servicios cloud aporta a las empresas la posibilidad de poder explorar nuevas soluciones con una inversión y riesgo mínimo.

Existen tres modelos de implementación en la nube:

- Nube pública: la nube pública es una infraestructura de nube compartida propiedad de un proveedor de nube, generalmente compañías muy grandes y conocidas globalmente (Amazon, Microsoft, Google) las cuales se encargan de su mantenimiento y gestión. Los principales beneficios de la nube pública son su escalabilidad bajo demanda y sus precios de pago por uso.

- Nube privada: este tipo de nube se ejecuta normalmente en la red de una empresa y está hospedada en un centro de datos dedicado para esa organización. La principal diferencia con un centro de datos corriente es la capacidad de desplegar y gestionar servicios a través de la red con un software de provisionamiento como OpenStack.

- Nube híbrida: como el nombre sugiere, el modelo de nube híbrida permite a las compañías integrar ambas soluciones de nube pública y privada. Con la nube híbrida, las organizaciones pueden aprovechar las bondades de cada modelo para potenciar la flexibilidad y escalabilidad, a la vez que protegen datos y operaciones confidenciales.

Aparte de los modelos de implementación, existen modelos de despliegue que categorizan los servicios dependiendo del nivel de gestión que tenga que realizar el usuario:

- Infraestructura como servicio (IaaS): es el modelo de despliegue menos gestionado. IaaS ofrece los conceptos más básicos de infraestructura para desplegar las aplicaciones tales como máquinas virtuales (instancias), disco, red, balanceadores… que pueden aprovisionarse a través de una API. Este modelo es lo más parecido a la la funcionalidad de un centro de datos tradicional en un entorno hospedado.

- Plataforma como servicio (PaaS): este modelo ofrece un entorno de desarrollo completo, eliminando la necesidad de que los desarrolladores se ocupen directamente de la capa de infraestructura al implementar o actualizar aplicaciones. Por ejemplo, una base de datos donde no hay que gestionar nada de configuración, actualizaciones o mantenimiento. Ejemplo: una base de datos

- Software como servicio (SaaS): las aplicaciones SaaS están diseñadas para usuarios finales, y mantienen detrás de escena todo el desarrollo y el aprovisionamiento de

infraestructura. Las aplicaciones SaaS ofrecen una amplia gama de funcionalidades en la nube: desde aplicaciones empresariales hasta plataformas de streaming de video. Ejemplo: Netflix

2.2 Contenedores

Históricamente, con el incremento en la capacidad computacional nacieron las máquinas virtuales (VM), diseñadas para ejecutar software en servidores físicos emulando un entorno aislado mediante un hypervisor, el cual se encarga de crear y gestionar las diferentes máquinas virtuales del entorno.
Cada máquina virtual se ejecuta con su propio sistema operativo, librerías, binarios y aplicaciones. Cada VM es independiente del hardware físico, pudiendo incluso tener otro sistema operativo.

Este concepto de virtualización ha cambiado en los últimos años con el uso de los contenedores, los cuales nacen con el objetivo de crear aplicaciones desacopladas, escalables y tolerantes a fallos con los microservicios, dejando atrás las aplicaciones monolíticas ejecutadas en la misma máquina.

Los contenedores son también una técnica de virtualización que no necesita de un hypervisor ni máquinas virtuales para ser ejecutados. En su lugar se ejecutan sobre el kernel del sistema operativo anfitrión, normalmente compartiendo librerías y binarios con el host. Los contenedores son parte de un tipo de virtualización llamada OS-level virtualization. En este caso, el kernel es el encargado de ejecutar los distintos contenedores y de aislarlos unos de otros, por lo que no hay un hypervisor ni un sistema operativo huésped.

Un contenedor es simplemente un proceso que utiliza estas características que se han mencionado en el kernel para tener una red, unos usuarios, un sistema de archivos, etc. La gran ventaja de los contenedores con respecto a las VMs es su tamaño y ligereza, ya que al compartir el sistema operativo con el sistema anfitrión no tiene que recrearlo por completo permitiendo ejecutar, migrar y reiniciar los contenedores de forma mucho más rápida.

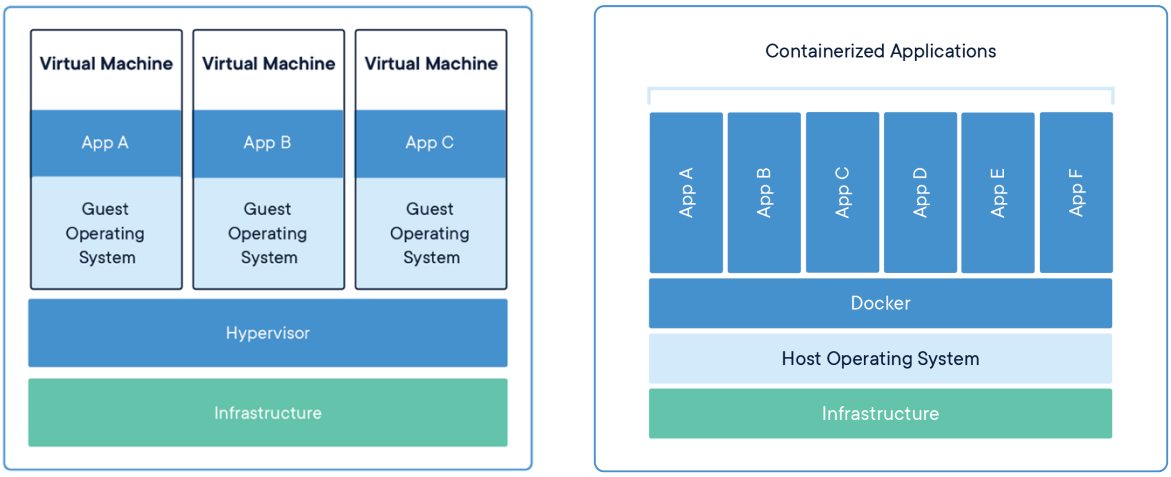

Figura 3. Máquinas virtuales vs contenedores

2.2.1 Docker

Docker[1] es un proyecto de código abierto que permite crear, probar e implementar aplicaciones de forma rápida. Docker empaqueta el software en contenedores[2] que incluyen todo lo necesario para que el software se ejecute, incluidas bibliotecas, herramientas de sistema, código y tiempo de ejecución.

Para ello, los contenedores de Docker utilizan las funcionales del kernel de Linux como

- CGroups: limita y aísla el uso de recursos (CPU, Memoria, disco… etc) para un conjunto de procesos
- Namespaces: los cambios realizados sobre un recurso son visibles únicamente por los procesos dentro de un mismo namespace.

La tecnología Docker no es la misma que los contenedores Linux tradicionales. Inicialmente, Docker se desarrolló a partir de LXC, la cual era útil como virtualización ligera, pero no ofrecía una buena experiencia de desarrollo y uso. Los contenedores de Linux tradicionales usan un sistema *init* para gestionar varios procesos. Esto implica que es posible ejecutar las aplicaciones completas como una sola. Por el contrario, Docker pretende que las aplicaciones se dividan en sus procesos individuales con el objetivo de desacoplar los servicios de las aplicaciones, de ahí nace el concepto de microservicio, donde cada funcionalidad (servicio) se separa y ejecuta en uno o varios contenedores, comunicándose con los demás mediante protocolo HTTP vía APIs.

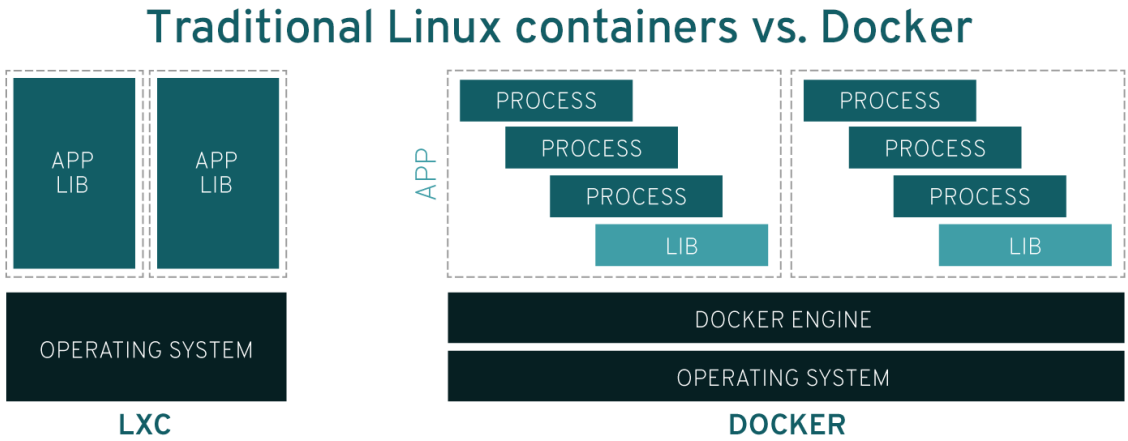

Figura 4. Contenedores LXC vs Docker

Docker no sólo aporta la capacidad de ejecutar contenedores, sino que facilita el proceso de creación, diseño, mantenimiento y gestión de imágenes de los contenedores entre otras cosas.

[1] Docker [2]
[1] ¿Qué es Docker? [7]
[2] What is a container [8]
[2] Lightweight linux containers for consistent development and deployment [16]

Además de proporcionar una API para comunicarse con las funcionalidades del kernel, la gran virtud de Docker es la facilidad con la que los usuarios pueden construir y compartir sus propios desarrollos en forma de imágenes Docker. Una imagen puede ser definida como un archivo comprimido que contiene todos los ficheros necesarios para que la aplicación se ejecute en el contenedor. Cada imagen está compuesta por una serie de capas donde se almacenan ficheros y librerías. Para crear una nueva imagen se puede partir de una imagen base, hacer las modificaciones que se estimen necesarias y guardar esas modificaciones como una nueva imagen. Para almacenar todas estas imágenes Docker hace uso de repositorios, los cuales pueden ser privados o públicos como DockerHub.

2.2.2 Orquestación de contenedores – Kubernetes[1]

Un orquestador es el encargado de gestionar el ciclo de vida de los contenedores de una aplicación. Un orquestador de contenedores generalmente ofrece funcionalidades como:

- Gestión de los nodos que componen el clúster permitiendo añadir o eliminarlos.
- Gestión de contenedores en ejecución, permitiendo reiniciar, parar, ejecutar…
- Servicio de descubrimiento para que los contenedores sean capaces de encontrar las rutas IP/DNS para comunicarse con otros contenedores que estén en ejecución.
- Servicios de balanceo de carga en los servicios desplegados
- Monitorización de los contenedores (CPU, memoria, almacenamiento…)
- Sondas continuas para comprobar que los contenedores siguen vivos (health checks).

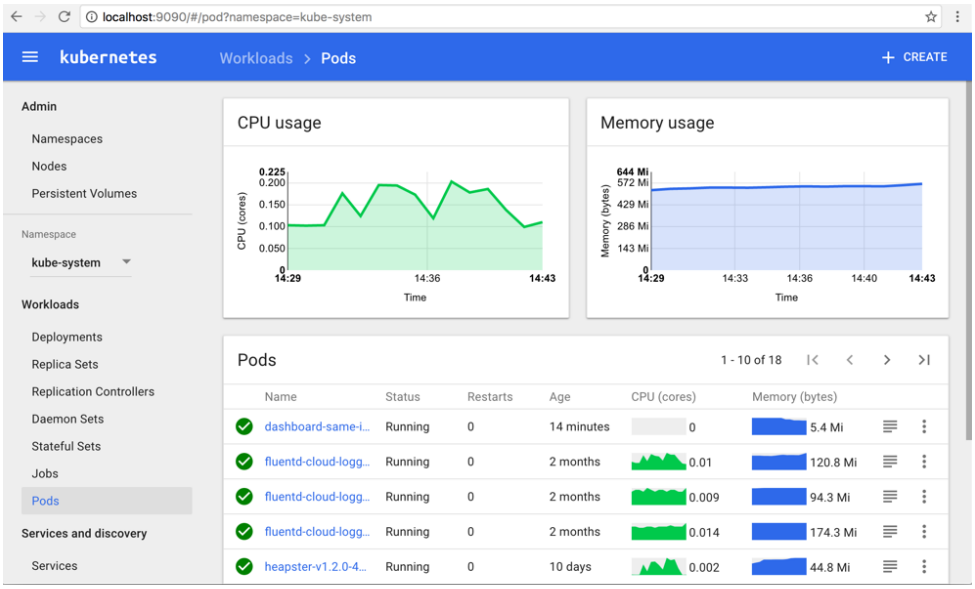

Figura 5. Consola de administración de Kubernetes

[1] ¿Qué es Kubernetes y cómo funciona? [9]

Un clúster de Kubernetes se compone de dos tipos de nodos: masters y esclavos. Por lo general los nodos master no ejecutan contenedores, son los encargados de coordinar el clúster, deciden cómo se distribuyen los contenedores entre los nodos, asegurando el número de réplicas y actualizando las aplicaciones de forma coordinada al desplegar nuevas versiones. La unidad mínima de ejecución en Kubernetes es el pod, que representa uno o más contenedores.
Un nodo master ejecuta los siguientes procesos[1]:

- API server: API usada para la comunicación con el clúster, por ejemplo, desde un cliente como kubectl.
- Scheduler: componente encargado de decidir donde se ejecuta cada contenedor.
- Controller manager: encargado de ejecutar controladores con el objetivo de asegurar que se cumple el estado de la aplicación. Existen diferentes tipos de controladores:
    - Node controller: comprueba el estado de los nodos físicos del clúster.
    - Replication controller: mantiene el numero correcto de pods para cada despliegue.
    - Endpoint controller: posibilita la conexión entre servicios y pods.
    - Service account & token controller: encargado de administrar crear cuentas y acceso a las APIs
- etcd: Base de datos clave-valor usada para guardar los datos y configuración del clúster.

Si el clúster de Kubernetes se despliega en un entorno cloud, existen más controladores para interactuar con los recursos del proveedor cloud:

- Node controller: controla los nodos físicos del clúster.
- Route controller: configura en la infraestructura las rutas entre los servicios del clúster
- Service controller: crea, actualiza y elimina balanceadores de carga
- Volume controller: crea, asocia, monta y formatea los volúmenes persistentes.

[1] Kubernetes components [18]

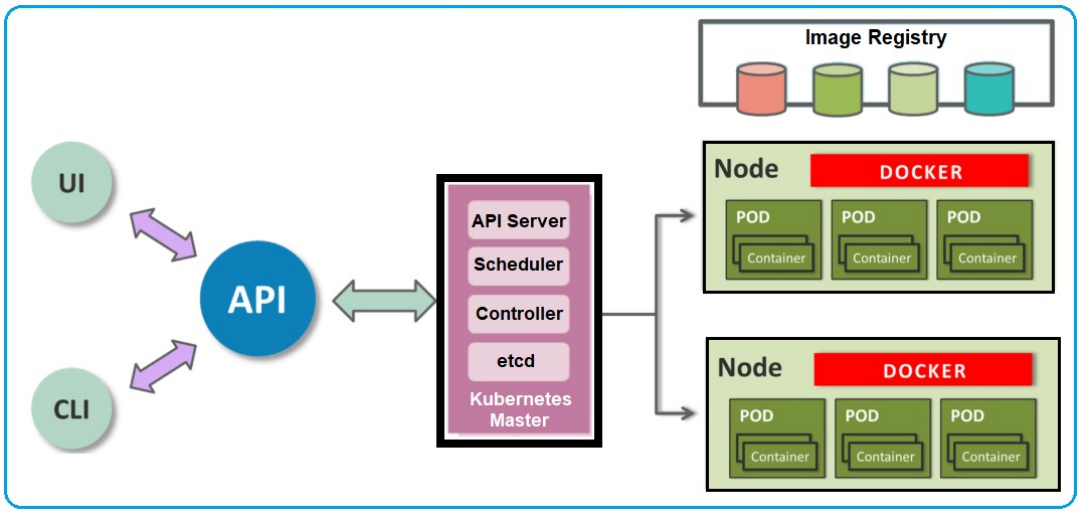

Figura 6. Arquitectura de Kubernetes

Los nodos esclavos son aquellos que ejecutan contenedores. Tienen tres elementos básicos.

- Software de ejecución de contenedores (Docker, rktlet o cri-o)
- kube-proxy: encargado de gestionar la red, así como las IPs asignadas a cada contenedor
- kubelet: componente más importante, encargado de asegurarse que todos los contenedores que deben ejecutarse en el nodo lo están haciendo. Si un contenedor falla o muere, kubelet lo reiniciará.

Por defecto Docker permite la conexión entre los contenedores únicamente si están dentro de la misma máquina. Para hacer posible la conexión entre contenedores de diferentes nodos, las máquinas necesitan su propia dirección IP, lo que implica que se debe coordinar cuidadosamente el uso y asignación de puertos usados por los contenedores. Para ello, Kubernetes define el término servicio, el cual es una abstracción que contiene un conjunto de pods y una política de cómo acceder a ellos. Los servicios tienen distintos modos de despliegue:

- ClusterIP: expone el servicio en una IP interna, es accesible desde dentro del clúster.
- Nodeport: expone el servicio en el puerto especificado para cada nodo.
- LoadBalancer: crea un balanceador de carga en el proveedor de cloud (si aplica) y asigna una IP fija externa al servicio. Los servicios ClústerIP Nodeport a los que el balanceador enruta son creados automáticamente.
- ExternalName: asocia el servicio a un nombre externo mediante un registro tipo CNAME.

2.3 Ingesta de datos en tiempo real – Kafka[1]

Apache Kafka es un sistema de almacenamiento publicador/suscriptor distribuido, particionado y replicado. Estas características, en consonancia con su rapidez en lecturas y escrituras lo convierten en una herramienta excelente para comunicar streams (flujos) de información que se generan a gran velocidad y que deben ser gestionados por uno o varias aplicaciones. Kafka es distribuido, se divide en varios nodos dentro de un clúster para su ejecución. Esta faceta, permite que sea escalable, pues está diseñado para crecer horizontalmente, añadiendo nodos de forma rápida.
Para aporta durabilidad y tolerancia a fallos, los mensajes son persistidos en el sistema de ficheros y replicados entre los nodos del clúster.

El modelo publicador/suscriptor es un patrón bajo la tipología de colas de mensajes, utilizado para la comunicación entre aplicaciones.
Existe un publicador o productor que genera los mensajes, los cuales son enviados al topic. Todos los consumidores suscritos a ese topic recibirán el mensaje.
La principal diferencia con las colas de mensajes reside en que ahí los consumidores deben obtener activamente los mensajes de la cola, mientras que en un topic se distribuyen. Además, un mensaje en una cola puede tener un único consumidor, mientras que, en los topics, todos los consumidores suscritos recibirán una copia del mensaje.

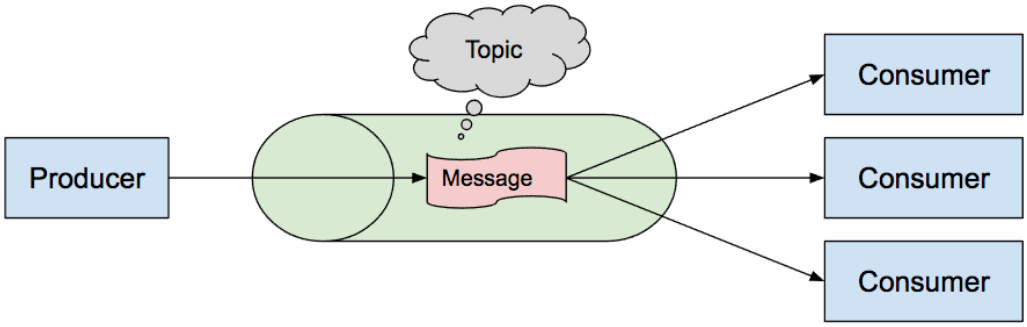

Figura 7. Patrón publicador/suscriptor

Este tipo de comunicación se utiliza principalmente para comunicación asíncrona. Se distinguen diferentes configuraciones:

- Tradicional: cada suscriptor está asociado a uno o varios topics, existen algunas variaciones.
  - Cada suscriptor escucha en un topic propio.
  - Cada suscriptor escucha en *n* topics independientes
  - Cada suscriptor escucha en *x* topics independientes e *y* topics compartidos
- Grupos de consumo: Los suscriptores se pueden agrupar por grupos, cada grupo está escuchando en un topic y sólo un miembro del grupo tendrá la capacidad de atender el mensaje.

[1] Primeros pasos con Kafka [14]

- Radio difusión: Todos los suscriptores que están escuchando el topic reciben el mensaje y cada uno es responsable de interpretarlo.

Kafka se compone de diferentes elementos que determinan la manera en que trata los mensajes. Cada topic se divide en una o más particiones, las cuales son secuencias inmutables de mensajes ordenados. Estas particiones puedes estar replicadas a su vez en diferentes nodos y brokers del clúster para facilitar la redundancia, escalabilidad y velocidad en la lectura permitiendo a los consumidores leer de forma paralela.
Cuando se replica una partición en varios brokers, se selecciona uno de ellos como líder, el cual recibe las peticiones de escritura por parte de los productores. En cada partición se escribe de forma independiente, cada vez que un mensaje es escrito, se le asigna un offset (identificador numérico ascendente).

Los topics en Kafka poseen varios parámetros de configuración que definen su comportamiento, como el número de particiones, que determina la cantidad de fragmentos en los que se divide un topic entre todos los brokers. Por otra parte, el factor de replicación es el número de copias de las particiones que se guardarán en el clúster. Con este parámetro se obtiene tolerancia a fallos y debe ser menor o igual al número de brokers (normalmente 3). También se puede configurar la retención en un topic con reglas que determinan si un mensaje caduca (por tiempo, tamaño, último mensaje producido con una clave determinada… etc).

2.4 Kops

Cualquier aplicación mínimamente compleja posee múltiples contendores encargados de dar funcionalidad, por ejemplo, uno para la base de datos, otro para el servidor web… etc. Durante las primeras etapas de desarrollo y test, tiene sentido desplegar todos los componentes de la aplicación en una sola máquina, pero a medida que la aplicación es desplegada en un entorno productivo, el enfoque monolítico se convierte en un punto único de fallo, ya que cualquier problema a nivel de host provocaría que la aplicación deje de funcionar.
Un clúster de Kubernetes generalmente se puede desplegar de varias formas dentro de un entorno cloud, desde servicios administrados donde no es necesario preocuparse por la infraestructura subyacente como EKS (servicio gestionado de Kubernetes en AWS), hasta un despliegue mucho más personalizable sobre máquinas virtuales.
Por lo tanto, si la elección es esta última, lo óptimo es desplegar la infraestructura en varias máquinas teniendo en cuenta entre otras cosas, la escalabilidad y alta disponibilidad de la solución, lo que involucra bastantes servicios y conocimientos acerca del proveedor cloud específico (red, balanceo de carga, máquinas virtuales, seguridad… etc). Es en el despliegue de infraestructura donde entra en juego Kops[1].

Kops es un proyecto oficial de Kubernetes que tiene como objetivo administrar, mantener y actualizar la infraestructura de clústeres productivos de Kubernetes desplegados sobre plataformas cloud. Actualmente, Kops está oficialmente soportado para AWS y en versión beta para GCP (Google Cloud Platform), OpenStack y DigitalOcean.

[1] What is Kops? [1]
[1] Kops repository. https://github.com/kubernetes/kops

Las características principales de Kops son las siguientes:

- Automatización del aprovisionamiento de la infraestructura subyacente para clústers de Kubernetes.
- Despliegue de los clústers en alta disponibilidad.
- Permite desplegar clústeres públicos o privados.
- Se basa en un modelo de sincronización de estado. Guarda el estado del clúster en un almacenamiento persistente de objetos para poder recrearlo a partir de cualquier versión anterior de manera consistente.
- Soporta addons de Kubernetes.
- La configuración se basa en ficheros YAML al igual que los despliegues en Kubernetes.
- Opción de hacer plantillas genéricas para desplegar clúster desde cero.
- Personalizable en su configuración, clústeres multi master, pudiendo elegir el numero de nodos, así como su tamaño y opciones de autoescalado.
- Kops puede manipular directamente la infraestructura o apoyarse de herramientas de infraestructura como código como Terraform o Cloudformation.

Por lo tanto, con el uso de una herramienta como Kops que se encarga de desplegar la infraestructura para albergar el clúster de Kubernetes, no es necesario conocer en profundidad el proveedor cloud como para desplegar la solución altamente disponible, escalable y tolerante a fallos, Kops se encarga de ello.

2.5 Fission

Fission[1] es un framework *open-source* para desplegar funciones serverless orientadas a eventos sobre Kubernetes. Esta solución Function as a Service (FaaS) opera a nivel de código, tanto Docker como Kubernetes son transparentes para los desarrolladores, los cuales tienen la posibilidad de codificar, desplegar y operar aplicaciones directamente en Kubernetes en cualquiera de los lenguajes soportados (Python, Node, Go, Bash…)
Para obtener un buen rendimiento, Fission mantiene contenedores "calientes" que contienen los componentes esenciales de la función. De esta manera, cuando una función es llamada por primera vez (cold start) uno de estos contenedores carga los componentes restantes y específicos para la función. El tiempo de cold start es del orden de 100 milisegundos.

[1] https://fission.io/ [11]

El flujo de desarrollo en Fission se divide en varios pasos:

1. Se desarrolla el código a ejecutar durante un corto período de tiempo.
2. Mapear la función a un trigger que desencadenará su ejecución como temporizadores, colas, peticiones HTTP, topics de Kafka... etc.
3. La función se despliega instantáneamente en el clúster, no hay que construir contenedores ni imágenes Docker.
4. Cuando un evento lanza la función, ésta se instancia con la arquitectura subyacente necesaria.
5. Si la función está en reposo sin recibir eventos durante el tiempo configurado, los recursos se liberan.

2.5.1 Arquitectura

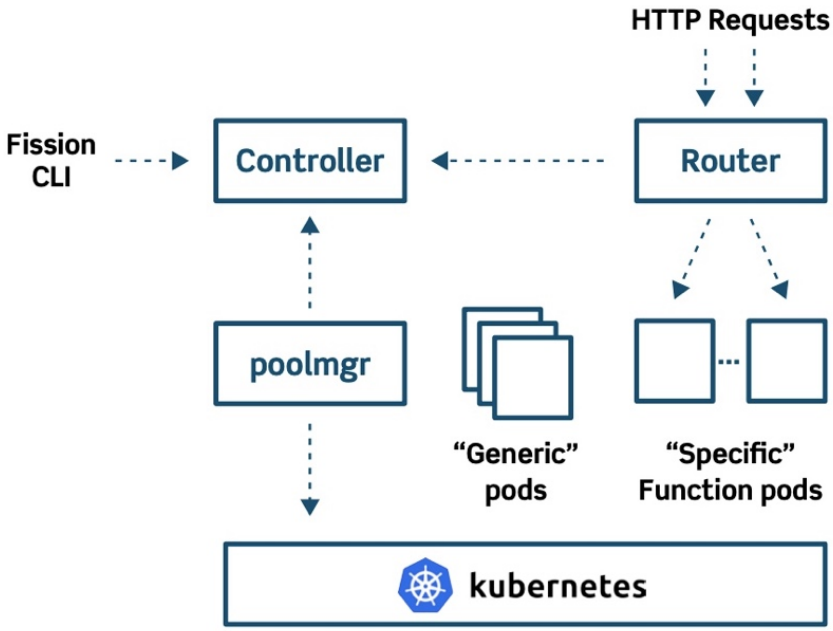

Figura 8. Arquitectura de Fission

Fission está diseñado como un conjunto de microservicios. El controller se encarga de revisar las funciones, rutas HTTP, triggers y las imágenes del entorno. También se encarga de exponer la API de Fission

El poolmgr controla los contenedores en reposo, así como las funciones ahí contenidas, matándolas cuando se alcanza el timeout.

El router se encarga de direccionar las peticiones HTTP a las funciones y actúa como un servicio de Kubernetes, de *LoadBalancer* o *NodePort*, dependiendo de donde esté el clúster hosteado. Cuando el router recibe una petición, consulta la caché para comprobar si ya está registrado el servicio al que hay que enrutar la petición, si no existe la entrada, busca la

función pertinente a la que asociar la petición y solicita una instancia, es decir, un pod libre al poolmgr para que cargue la función. Una vez la función es levantada, el router cachea sus propiedades para enrutar las futuras peticiones.

Desarrollar aplicaciones complejas mediante el paradigma serverless, requiere la composición e interacción de los diferentes elementos. Para lidiar con esta tarea, Fission WorkFlows provee un framework que permite definir flujos y secuencias de tareas compuestas por funciones, pudiendo orquestarlas y sin necesidad de preocuparse por la configuración de red o colas.

Los logs en Fission se controlan vía Fluentd e InfluxDB. El primero de ellos se encarga de recolectar los logs y enviarlos a la base de datos (InfluxDB) donde son guardados. Gracias a una base de datos como esta orientada a guardar datos de series temporales, es muy sencillo obtener métricas, filtrar, agregar y hacer análisis de los datos.

En cuanto a monitorización, Fission lo hace mediante Prometheus, una herramienta open source centrada en monitorización sobre Kubernetes. Con Prometheus se muestran métricas de las funciones como el número de invocaciones, el número de errores, el tiempo de ejecución, tamaños de respuesta, cold starts… etc.

2.6 Elasticsearch y Kibana

Para aportar a la solución con la habilidad de poder consultar los datos en tiempo real pudiendo realizar búsquedas y filtros sobre el texto generado y mostrar esos datos en útiles dashboards con el estado de cada dispositivo, se ha optado por utilizar Elaticsearch en consonancia de Kibana.

2.6.1 Elasticsearch

Elasticsearch[1] es un motor de búsqueda y análisis de texto, distribuible y escalable. Está pensado para realizar búsquedas en tiempo real sobre grandes cantidades de datos. Este motor, es accesible a través de una extensa API desde donde es posible invocar búsquedas complejas de forma extremadamente rápida. Entre sus principales características, destaca su orientación a documentos JSON, no utiliza esquemas, aunque si es necesario se pueden definir. A su vez, es distribuido, lo que implica que los nodos se pueden repartir en diferentes máquinas físicas en configuraciones escalables en función de la demanda de forma horizontal y altamente disponible. Es posible realizar búsquedas tanto estructuradas como no estructuradas.

[1] ¿Qué es y cómo funciona Elasticsearch? [5]
[1] ElasticSearch: An advanced and quick search technique to handle voluminous data [6

Las principales ventajas de Elasticsearch son las siguientes:

- Velocidad: Elasticsearch encuentra de manera muy rápida resultados a complejas búsquedas en grandes volúmenes de información basándose en índices invertidos distribuidos.

- API sencilla: La interfaz HTTP que expone Elasticsearch es muy potente, además de permitir búsquedas no estructuradas (sin esquema) lo que facilita la búsqueda, indexación y consulta de los datos.

- Actualizaciones de índices: Las actualizaciones de los índices que componen el motor de búsqueda son realmente rápidas. Acciones como añadir un documento recién indexado al índice suele tardar menos de un segundo, pudiendo manejar aplicaciones que necesitan un tiempo de respuesta muy bajo.

- Soporte amplio para gran cantidad de lenguajes de programación como Java, Python, Javascript, NodeJS … etc

Elasticsearch no trabaja como software de manera independiente, normalmente va acompañado de un motor de recopilación de logs llamado Logstash y una plataforma de visualización y análisis llamado Kibana. Estos tres productos están diseñados para trabajar de manera conjunta en una solución llamada Elastick Stack (anteriormente ELK).

Elasticsearch se despliega en modo clúster, que al igual que en Kafka, se tienen un conjunto de nodos que mantienen toda la información distribuida e indexada. Un clúster debe tener al menos 3 nodos y un número impar de ellos para evitar corrupción de datos si uno de ellos cae, y si es el líder, los otros nodos deciden el nuevo líder mediante *quorum.*

Un index (índice) es una colección de documentos que tienen características similares. Los índices están identificados por un nombre, el cual es usado a la hora de indexar, buscar, actualizar y borrar. Cuando los índices sobrepasan un tamaño concreto, Elasticsearch permite realizar *sharding* para no decaer en rendimiento escalando de manera horizontal, esto es, partir los índices en diferentes nodos paralelizando las peticiones que se hagan dependiendo del nodo en el que se encuentre la información. La replicación permite que los índices puedan ser reconstruidos si un nodo cae.

2.6.2 Kibana

Kibana[1] es una herramienta *open-source* perteneciente a Elastic, que sobre los datos que se encuentran indexados en ElasticSearch, permite visualizar y explorar datos provenientes de logs, series temporales o monitorizar aplicaciones entre otras cosas. Con elementos como histogramas, diagramas de calor, soporte nativo para datos geospaciales, hace sencillo analizar y obtener conclusiones de los datos recolectados.

[1] Introducción a Kibana [4]

Algunas de las características principales:

- Descubrimiento de datos: Kibana detecta los datos en elasticsearch y permite crear índices en base a patrones de texto.

- Gráficas interactivas e intuitivas. Es posible definir dinámicamente la ventana de tiempo sobre la que se visualizan los datos, hacer zoom sobre conjuntos específicos de datos… etc.

- Análisis geospacial de forma nativa con coordenadas como tipo de dato.

- Consola de desarrollador incorporada para hacer peticiones directamente a Elasticsearch el estilo cURL

- Dashbards preconfigurados para fuentes de datos comunes sin necesidad de tener que configurar nada más que la conexión.

3 Diseño y desarrollo

3.1 Visión general

La aplicación diseñada se encarga del flujo completo de los datos generados por los dispositivos IoT, desde la generación de los eventos hasta la visualización de estos.
Este flujo se distingue en 4 etapas:

Figura 9. Arquitectura general de la aplicación

3.1.1 Generación de eventos

La consola de simulación de eventos es una interfaz web que permite crear y simular miles de dispositivos sin necesidad de administrar dispositivos físicos ni diseñar scripts personalizados.
Este simulador se basa en una arquitectura de microservicios, se expone una API a través de API Gateway que se encargar de llevar a cabo la lógica de la aplicación sobre los dispositivos virtuales invocando a las funciones lambda pertinentes para cada servicio.

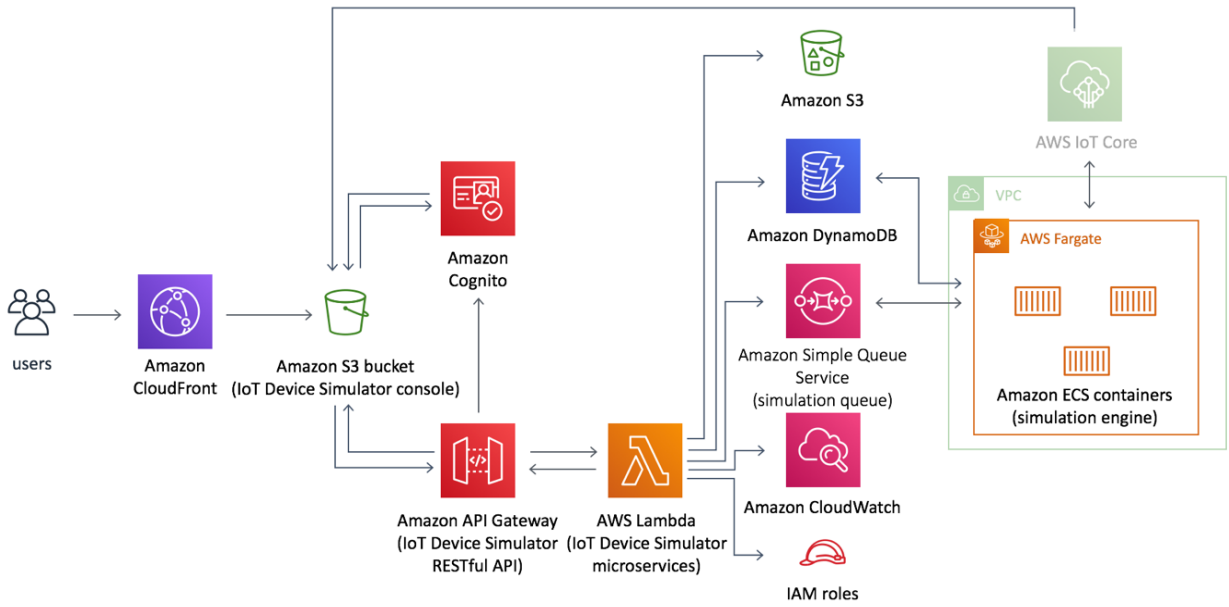

Figura 10. Simulador de eventos

Cuando una solicitud de simulación es recibida, el microservicio *dispositivo* envía dicha petición de simulación a una cola SQS, la cual es constantemente consultada por contenedores desplegados en Fargate, que, ante la necesidad de comenzar una nueva simulación, comienzan a interactuar con un endpoint de AWS IoT Core publicando la información simulada. Cuando la simulación finaliza, las métricas son guardadas en DynamoDB (base de datos no relacional), siendo posteriormente presentadas en la interfaz gráfica.

La interfaz web está mayormente alojada en S3, ya que gran parte del contenido es estático. Este contenido es expuesto a Internet mediante Cloudfront, el servicio de CDN (red de distribución de contenido) que ofrece AWS, teniendo la web disponible de forma global con una latencia muy baja.

Figura 11. Simulador de eventos. Pantalla principal de simulaciones.

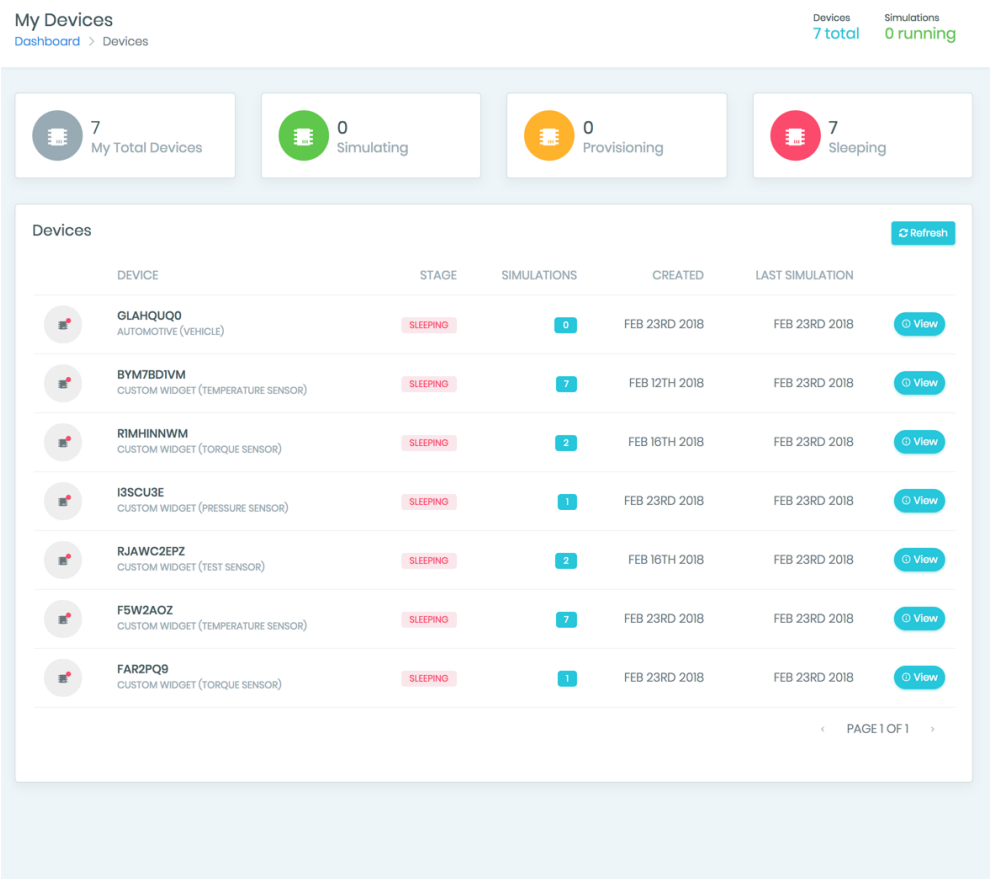

Figura 12. Simulador de eventos. Pantalla de dispositivos creados.

3.1.2 Ingesta y procesamiento

Una vez los eventos son generados, se envían a IoT core. AWS IoT core es un servicio totalmente administrado que actúa como un punto central y permite a todos los dispositivos conectados interactuar de manera fácil, fiable y segura con otros dispositivos y aplicaciones en la nube. Los dispositivos se pueden comunicar mediante varios protocolos (HTTP, MQTT o WebSockets). IoT core ofrece la posibilidad de configurar autenticación y cifrado integral en todos los puntos de conexión con el objetivo de que los datos siempre se intercambien con de forma segura entre entidades verídicas y probadas. También se pueden asignar permisos granularizados a cada dispositivo para cumplir el paradigma de mínimos privilegios.

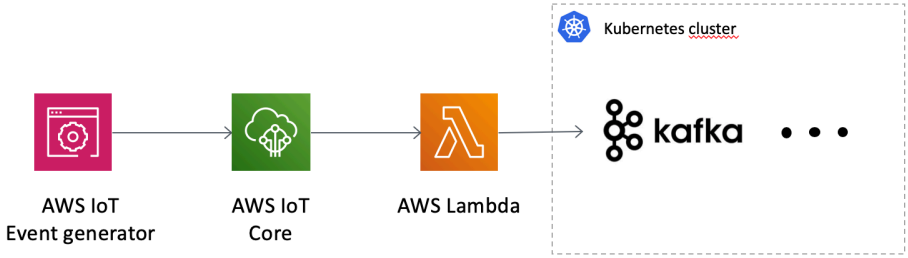

Figura 13. Ingesta en Kafka desde el generador de eventos

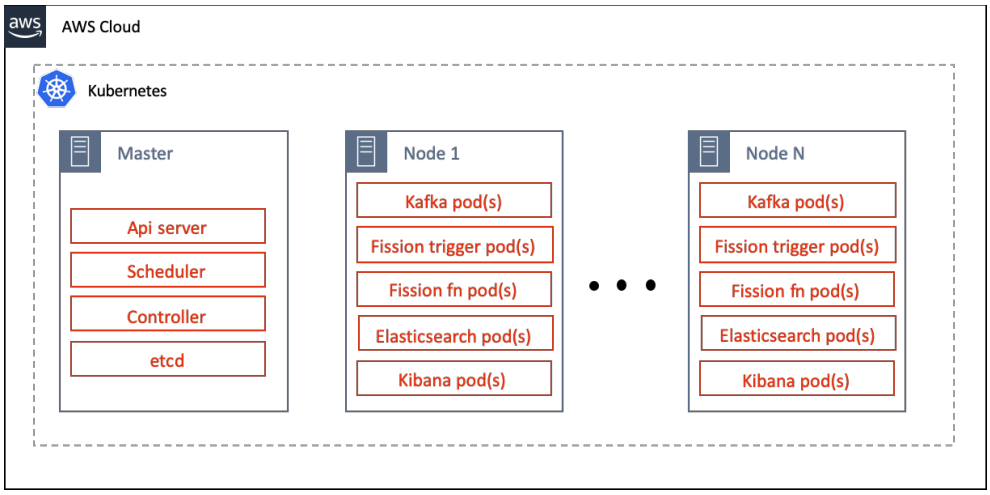

Figura 14. Distribución del clúster de Kubernetes

Dado que la intención es tener totalmente desacoplado el sistema de generación de eventos del resto de la aplicación, es necesario un punto de unión entre ambos. En este caso, se ha desarrollado una función lambda que se dispara con cada evento que llega a IoT core.
Esta Lambda recibe el evento mediante HTTP, lo procesa mínimamente cambiando de formato algún campo como la localización o el timestamp. Posteriormente, el evento es enviado al topic de Kafka mediante MQTT desde la Lambda, la cual actúa como productor. Para poder comunicar la función lambda con Kafka y poder insertar los mensajes, se ha optado por usar el SDK *Kafka-python*, empaquetando dichas librerías junto al código.

Una vez los mensajes han sido depositados en el topic de Kafka, es necesario configurar todo el entorno de Fission para que los procese. El primer paso, indicar a Fission el endpoint del broker de Kafka, en este caso, como Kafka está desplegado en Kubernetes en modo *LoadBalancer,* este endpoint corresponde al DNS del balanceador de AWS.
El siguiente paso es crear la función que va a procesar los mensajes, esto es realmente sencillo con el CLI que Fission ofrece:

```
fission fn create --name kafka-to-elasticsearch --env pythonenv --src
elasticsearch.zip  --entrypoint "elasticsearch_raw.main" --buildcmd
"./build.sh" --minscale 1 --maxscale 3
```

Entre los parámetros que recibe el comando para crear una función, cabe destacar el archivo .zip que contiene el código, un fichero requirements.txt con las librerías necesarias como el SDK de elasticsearch y un script bash (build.sh) con el comando que instala las librerías en el contenedor docker en el que se ejecutará la función. Para el autoescalado de la función se define el número mínimo y máximo de pods que puede abarcar. El porcentaje de uso de CPU por defecto es 80%, Fission ajustará el número de pods, añadiendo o eliminando para ceñirse a esa métrica. Este porcentaje es configurable al crear la función con el parámetro *targetcpu.*

Una vez creada la función, se puede crear y asociar a la misma el trigger que escucha en el topic de Kafka como consumidor y que será el encargado de ejecutar la función con la llegada de cada mensaje.

```
fission mqt create --name kafka-trigger --function elasticsearch --
mqtype kafka --topic input --resptopic output --errortopic error
```

Los parámetros necesarios se constituyen por el nombre del trigger, la función a ejecutar, el topic donde el trigger va a escuchar y dos topics adicionales donde la función escribirá el resultado o los errores de la ejecución.

3.1.3 Visualización

Para concluir el ciclo de vida de los mensajes y poder sacar el verdadero valor de los datos, en necesario crear gráficos que representen el comportamiento de los dispositivos.
Una vez los eventos han sido insertados e indexados en Elasticsearch, se configura Kibana para que analice el índice y obtenga los tipos de datos de las variables dentro de los eventos.
Los dispositivos simulados para este trabajo son vehículos, por lo que se poseen gran cantidad de métricas. Cada vehículo posee las siguientes métricas:

```
{
   "timestamp":  "2019-06-15T11:39:31.639737Z",
   "vin": "BSTZ9IEK39HHYZB6I",
   "vehicle_speed": 118.4,
   "odometer": 42254,
   "engine_speed": 3117.2,
   "brake": 0,
   "oil_temp": 80.3,
   "parking_brake_status": "false",
   "acelerator_position": 43,
   "steering_wheel_angle": 0,
   "torque": 341,
   "gear": 5,
   "location":{
      "lat": 38.95618,
      "lon": -77.39835
   },
   "fuel_level": 76.8,
   "fuel_consumed_since_start": 0.453435,
   "acceleration": 0.4982,
   "ignition_status": "run"
}
```

- timestamp: hora en la que se produce el evento.
- vin: matricula del vehículo.
- vehicle_speed: velocidad del vehículo en km/h en el instante de tiempo.
- odometer: cantidad de kilómetros recorridos por el vehículo en total.
- engine_speed: revoluciones del motor.
- brake: 1 si el freno está activado, 0 si no.
- oil_temp: temperatura de aceite.
- parking_brake_status: *true* si el freno de mano está activo y *false* en caso contrario.
- acelerator_ position: porcentaje de acelerador.
- steering_wheel_angle: ángulo del volante.
- torque: par motor en el instante de tiempo.
- gear: marcha actual.
- location: coordenadas del vehículo en el instante de tiempo.
- fuel_level: nivel de combustible.
- fuel_consumed_since_start: combustible gastado desde el inicio del trayecto.
- acceleration: tasa de aceleración en m/s^2
- ignition status: indica si el vehículo está en marcha o parado.

Un dashboard en Kibana constituye un conjunto de visualizaciones[1], búsquedas, mapas… etc normalmente actualizándose *quasi* en tiempo real. Para crear las visualizaciones que compondrán el dashboard, es necesario tener los datos indexados en Elasticsearch. Kibana ofrece gran cantidad de gráficos (de barras, lineal, tablas, deslizadores…), una vez elegido el deseado, es necesario seleccionar los campos indexados que se quieren representar, por ejemplo, el timestamp en el eje horizontal y la velocidad de los vehículos en el eje vertical para observar como varía la velocidad en función del tiempo para cada vehículo.

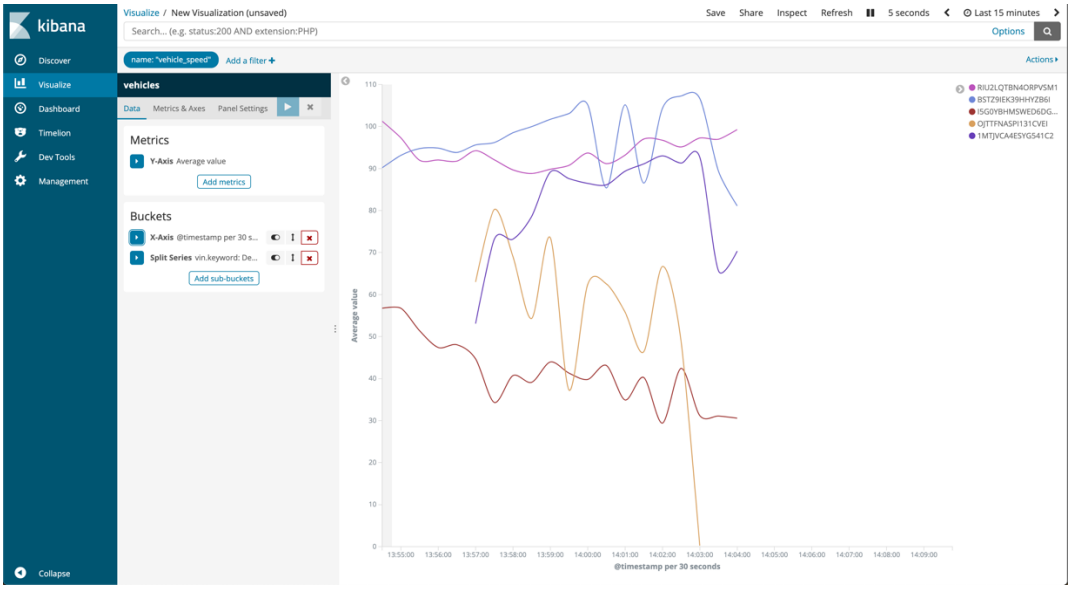

Figura 15. Visualización en Kibana. Velocidad en función del tiempo.

Una vez se han creado todas las visualizaciones deseadas, se crea el dashboard colocando estas visualizaciones en el espacio y tamaño deseado.

[1] Creating a visualization [10]

4 Pruebas y resultados

Para comprobar el correcto funcionamiento de la aplicación, se ha realizado un primer test extremo a extremo. Esto implica el flujo completo de los datos, desde que se generan hasta que son visualizados, atravesando todos los puntos intermedios de captura y procesamiento.

En primera instancia, se ha realizado una prueba con una pequeña muestra, generando cinco vehículos y se simulando una ejecución durante treinta minutos.
Una posible primera visualización es representar la cantidad de eventos recibidos a lo largo de la simulación y la velocidad media de los

Figura 16. Simulación 5 vehículos. Eventos a lo largo del tiempo y velocidad media

Con esta pequeña prueba se puede ver cómo los eventos han llegado al clúster de Kubernetes y se han procesado correctamente, desde Kafka hasta Elasticsearch, que ha indexado los mensajes e inferido su esquema haciendo posible realizar las agregaciones para visualizarlas en Kibana.

Posteriormente se ha creado un dashboard con diferentes tipos de gráficos, dependiendo de las métricas representadas:

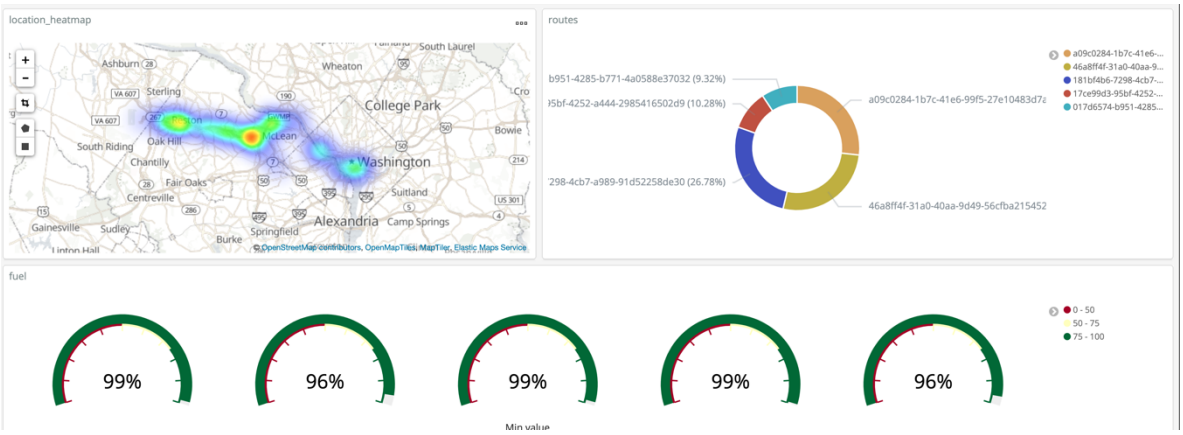

Figura 17. Dashboard general

La primera visualización es un mapa de calor, resaltando las zonas geográficas por las que más tráfico ha habido. El simulador crea eventos con coordenadas GPS en carreteras cercanas a Washington, Kibana es capaz de interpretar el tipo de dato localización de forma nativa, por lo que representar los datos en el mapa es realmente sencillo.

La segunda visualización muestra las carreteras más transitadas por porcentaje en un gráfico circular. En este caso, el identificador de las rutas no es muy representativo, pero el tipo de gráfico puede ser el más representativo para ese tipo de dato.

Finalmente, se muestra el combustible para los diferentes vehículos. El gráfico cambia de color dependiendo del porcentaje configurado.

Además, también es posible añadir filtros bajo el mismo dashboard para mostrar únicamente ciertos datos, por ejemplo, de ciertos dispositivos o una métrica en concreto. Aplicando los filtros a los vehículos, se pueden disgregar por matrícula:

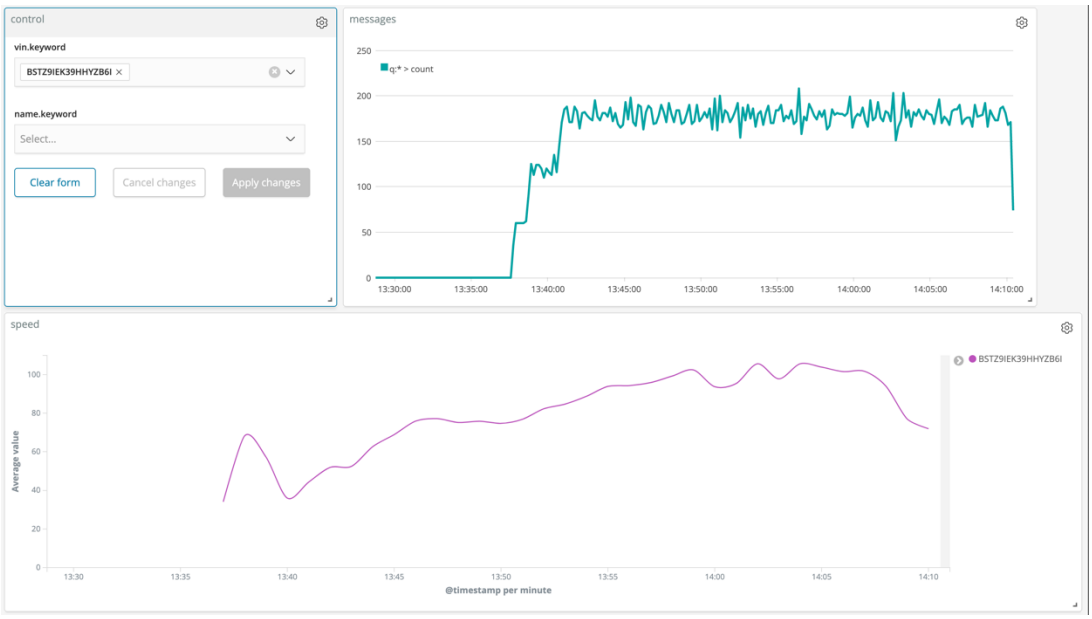

Figura 18. Dashboard con filtros. Eventos recibidos y velocidad media.

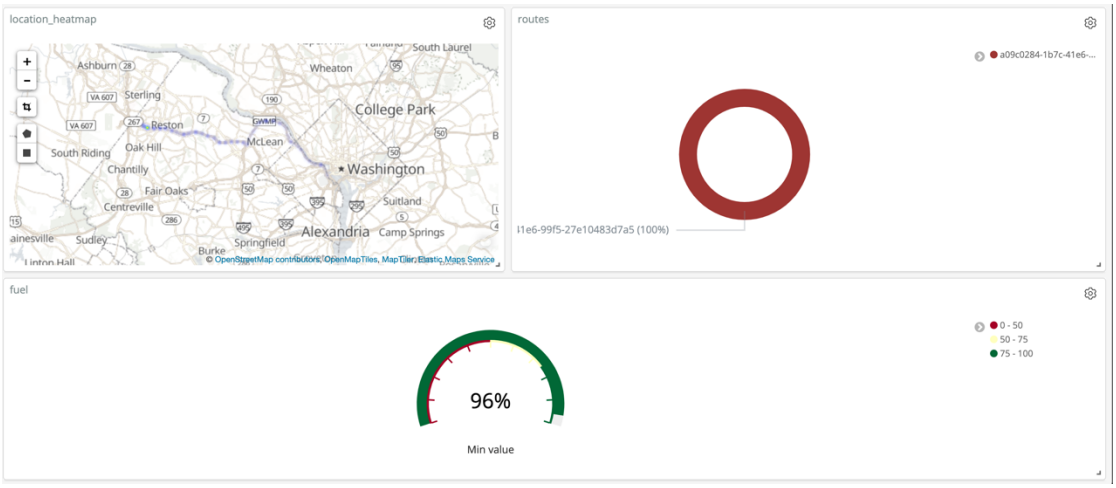

Figura 19. Dashboard con filtros. Localizacion y combustible.

Con el objetivo de probar más a fondo la plataforma y la aplicación, se han realizado unas pruebas de rendimiento, que a pesar de no haber podido finalizarlas por razones de presupuesto, son válidas para sacar algunas conclusiones.
Uno de los desafíos que surgen al realizar pruebas de rendimiento como estas, es la manera de generar los eventos, pues es necesaria una tasa muy alta para estresar una arquitectura con autoescalado. En este aspecto, los eventos se generaron de dos formas diferentes:

- Mediante el simulador de eventos mencionado anteriormente, creando 500 dispositivos, que al enviar un evento cada 2 segundos, supone una tasa de 250 mensajes por segundo.

- Utilizando Apache JMeter en un ordenador local enviando los mensajes directamente al topic. Para enviar los eventos mediante el protocolo MQTT se utilizó el plugin *Pepper Box*. Con este método se consiguió una tasa 500 eventos por segundo, el factor limitante fue el hardware.

Tras esta prueba, el sistema aguantó la carga descrita durante 1 hora sin problemas, ya que no es un gran volumen de datos. El principal problema reside en que dicha prueba generó una cantidad de costes considerables, por lo que repetirla fue inviable.
Para aumentar la tasa de eventos desde Apache JMeter la solución pasa por levantar múltiples instancias spot y generar todos los eventos posibles de manera concurrente. Este tipo de instancias son mucho más baratas que al ejecutarlas en pago por uso, hasta un 90% más baratas, a expensas de que AWS las elimine si el precio de puja establecido está por debajo del precio actual de mercado.

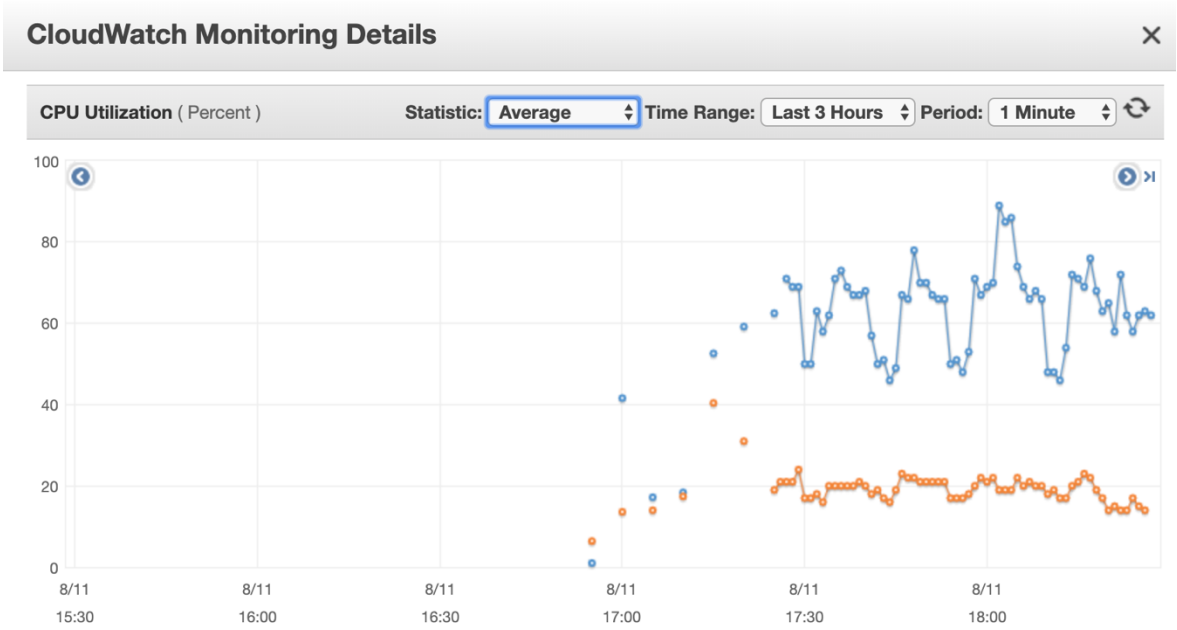

Figura 20. Monitorización de instancias en prueba de rendimiento.

Tras las pruebas de rendimiento el master de Kubernetes estuvo en 20% de CPU mientras que los 3 nodos estuvieron en torno al 60%. La configuración de los nodos era de 2 vCPU y 4GB de RAM.

5 Conclusiones y trabajo futuro

5.1 Conclusiones

Teniendo en mente el objetivo principal del trabajo, el cual es acabar con una arquitectura base agnóstica de la infraestructura física sobre la que desplegar una aplicación IoT que permita el análisis de los datos generados por los dispositivos, se puede concluir que los componentes escogidos y su configuración son los adecuados para este caso de uso. Un aspecto importante era comprobar y determinar cómo funcionan ciertos productos como Kafka sobre Kubernetes, ya que originalmente se desarrolló para ser desplegado de forma individual y no formando parte de un conjunto contenedores, pero haciendo uso de la gran cantidad de posibilidades que ofrece Kuberetes como los despliegues de statefulsets, los cuales ofrecen identificadores estables y únicos para componentes como red, almacenamiento, despliegues ordenados… etc, es posible desplegar aplicaciones que deban llevar el control de todas sus partes.
A falta de realizar pruebas en mayor profundidad para determinar si esta solución sería viable para un caso de uso real, se puede determinar que el resultado del trabajo es positivo en términos generales.

5.2 Trabajo futuro

Ante un proyecto tan interesante y amplio como es este que incluye gran cantidad de campos sobre los que profundizar como el IoT, procesamiento de eventos en tiempo real o arquitectura cloud, se pueden hacer grandes mejoras para convertir esta solución en un producto realmente potente.
Una mejora importante que realizar con la mente puesta en llevar este proyecto a una empresa y al mundo real, es proveer a la solución de un ciclo de integración y despliegue continuo, de manera que el desarrollo y las pruebas del código se integren con la infraestructura, pudiendo desplegar de forma automatizada cualquier cambio que se realice a nivel de aplicación en diferentes entornos. Para ello se pueden implementar herramientas como Jenkins, Gitlab o Codepipeline. Además, es una buena práctica desplegar la infraestructura como código teniéndola codificada en plantillas y obteniendo el beneficio directo de poder replicar toda la solución con su correcta configuración con un solo click en cuestión de minutos. Para ello se pueden utilizar herramientas como Terraform, Cloudformation, Ansible, Chef o Puppet.

Otra opción interesante, es dotar a los datos de más valor, no sólo analizar hecho pasados sino predecir eventos futuros. Con el uso de machine learning orientado a series temporales se puede predecir los acontecimientos que ocurrirán en base al pasado. En este ejemplo concreto, se podrían predecir accidentes, atascos o el momento en el que un vehículo necesita mantenimiento.

Finalmente, un entorno de este estilo debe ser rigurosamente probado con gran profundidad, por lo que otro punto importante es realizar un plan de pruebas e integración para determinar la viabilidad y potencia de la solución.

Referencias

[1] Adam Hawkins (2017). What is Kops? Kubernetes Operations with Kops [Online]. Available: https://cloudacademy.com/blog/kubernetes-operations-with-kops/

[2] Anderson, C. (2015). Docker [software engineering]. IEEE Software, 32(3), 102-c3. Available: https://www.adictosaltrabajo.com/2014/10/13/kafka-logs/

[3] Bill Ward (2018). Installing Kafka Docker on Kubernetes [Online]. Available: https://dzone.com/articles/ultimate-guide-to-installing-kafka-docker-on-kuber

[4] Chabir Atrahouch (2015). Introducción a Kibana [Online]. Available: https://www.adictosaltrabajo.com/2015/12/27/introduccion-a-kibana/

[5] DavidOchobits (2019). ¿Qué es y cómo funciona Elasticsearch? [Online]. Available: https://www.ochobitshacenunbyte.com/2018/08/28/que-es-y-como-funciona-elasticsearch/

[6] Divya, M. S., & Goyal, S. K. (2013). ElasticSearch: An advanced and quick search technique to handle voluminous data. Compusoft, 2(6), 171.

[7] Docker: ¿Qué es Docker? [Online]. Avalilable: https://www.redhat.com/es/topics/containers/what-is-docker

[8] Docker: What is a container? [Online]. Available: https://www.docker.com/resources/what-container

[9] Eduard Tomás, (2019). ¿Qué es Kubernetes y cómo funciona? [Online]. Available: https://www.campusmvp.es/recursos/post/que-es-kubernetes-y-como-funciona.aspx

[10] Elastic (2019). Creating a visualization [Online]. Available: https://www.elastic.co/guide/en/kibana/current/createvis.html

[11] Fission.io [Online]. Available: https://fission.io/

[12] Github. (2019). Kops original repository [Online]. Available: https://github.com/kubernetes/kops

[13] Imesh Gunaratne (2018). Deploying Kubernets in AWS [Online]. Available: https://medium.com/containermind/how-to-create-a-kubernetes-cluster-on-aws-in-few-minutes-89dda10354f4

[14] Juan Alonso Ramos (2014) Primeros pasos con Apache Kafka [Online].

[15] Karimi, K., & Atkinson, G. (2013). What the Internet of Things (IoT) needs to become a reality. White Paper, FreeScale and ARM, 1-16.

[16] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239), 2.

[17] Rinu Gour (2019). Apache Kafka Load Testing Using JMeter [Online]. Available: https://dzone.com/articles/apache-kafka-load-testing-using-jmeter

[18] Tim Bannister, (2019). Kubernetes components [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/

[19] Vamsi Chemitiganti (2018). What is Serverless? Fission framework [Online]. Available: https://hackernoon.com/what-is-serverless-part-4-fission-an-open-source-serverless-framework-for-kubernetes-7f025517774a

[20] Van Kranenburg, R., & Bassi, A. (2012). IoT challenges. Communications in Mobile Computing, 1(1), 9.
[21] Ventajas IoT https://www.master-internet-of-things.com/ventajas-desventajas-del-uso-iot/
[22] Víctor Madrid (2018). Aprendiendo Apache Kafka [Online]. Available: https://enmilocalfunciona.io/aprendiendo-apache-kafka-parte-1/

El repositorio donde se encuentra el código y configuraciones desarrolladas es https://gitlab.com/agilhernan/tfm

Anexos

A. Instalar Kops y desplegar un cluster en AWS

1. Instalar AWS CLI

En OSX utilizando Homebrew
```
brew install awscli
```

Linux
```
pip install awscli --upgrade --user
awscli version: 1.6.5
```

2. Instalar Kops

En OSX utilizando Homebrew
```
brew install kops
```

Linux
```
curl -LO https://github.com/kubernetes/kops/releases/download/$(curl -s
https://api.github.com/repos/kubernetes/kops/releases/latest | grep
tag_name | cut -d '"' -f 4)/kops-linux-amd64
chmod +x kops-linux-amd64
sudo mv kops-linux-amd64 /usr/local/bin/kops
kops version: 1.9.0
```

El usuario debe tener los siguientes permisos en AWS (IAM) para desplegar el cluster
```
AmazonEC2FullAccess
AmazonRoute53FullAccess
AmazonS3FullAccess
AmazonVPCFullAccess
```

Configurar AWS CLI con las credenciales obtenidas de IAM
```
aws configure

AWS Access Key ID [None]: AccessKeyValue
AWS Secret Access Key [None]: SecretAccessKeyValue
Default region name [None]: eu-west-1
Default output format: [None]
```

Create S3 bucket for storing kops state
```
bucket_name=xxxxx
aws s3api create-bucket --bucket ${bucket_name} --region eu-west-1 --
create-bucket-configuration LocationConstraint=eu-west-1
```

Enable versioning in the bucket
```
aws s3api put-bucket-versioning --bucket ${bucket_name} --versioning-
configuration Status=Enabled
```

Set K8s cluster name and set S3 URL in env variables
```
export KOPS_CLUSTER_NAME=xxx.k8s.local
export KOPS_STATE_STORE=s3://${bucket_name}
```

Define cluster
```
kops create cluster --help
```

```
kops create cluster --node-count=2 --node-size=t2.micro --master-
size=t2.micro --zones=eu-west-1a --dns private --
name=${KOPS_CLUSTER_NAME}

export AWS_ACCESS_KEY=AccessKeyValue
export AWS_SECRET_KEY=SecretAccessKeyValue
```

3. Revisar definición del cluster a provisionar

```
kops edit cluster --name ${KOPS_CLUSTER_NAME}
```

4. Crear cluster

```
kops update cluster --name ${KOPS_CLUSTER_NAME} –yes
```

5. Obtener estado hasta que el cluster esté creado

```
kops validate clúster
```

6. Desplegar Kubernetes Dashboardd para poder ver el estado de los componentes del clúster

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.1/src/deploy
/recommended/kubernetes-dashboard.yaml
```

7. Obtener token de administrador para hacer login en la consola

```
kops get secrets kube --type secret -oplaintext
```

8. Acceder al dashboard

```
kubectl proxy
http://localhost:8001/api/v1/namespaces/kube-
system/services/https:kubernetes-dashboard:/proxy/
```

B. Arrancar y parar el cluster

1. Arrancar

Los nodos del clúster se escalan al número deseado.

```
export KOPS_STATE_STORE=s3://kops-state-store-agil
kops edit ig nodes

#get master name
kops get ig

#scale master to desired instances
kops edit ig master-xxx

kops update cluster --yes
kops rolling-update cluster
```

2. Parar

Los nodos del clúster se escalan a 0 para forzar la eliminación de las instancias. El estado del clúster será preservado en un bucket s3 para poder restaurarlo posteriormente.

```
kops edit ig nodes

#get master name
kops get ig
```

```
#scale master to 0 instances
kops edit ig --name master-xxx
kops update cluster --yes
kops rolling-update cluster
```

C. Instalar Fission en Kubernetes

```
helm install --name fission-all --namespace fission
https://github.com/fission/fission/releases/download/1.1.0/fission-all-
1.1.0.tgz -f values.yaml
```

En el fichero values.yaml se debe definir el endpoint público del boker de Kafka para poder definir los trigger