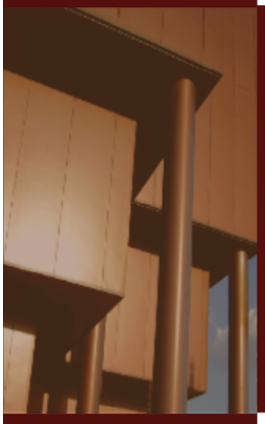
Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Agregador de datos para IoT nativo en la nube

Autor: Jose Fuentes Castillo
Tutor: Clara Isabel Luján Martínez



Dpto. de Ingeniería Electrónica
Escuela Técnica de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Agregador de datos para IoT nativo en la nube

Autor:
Jose Fuentes Castillo

Tutor:
Clara Isabel Luján Martínez
Profesor titular

Dpto. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2018

Proyecto Fin de Carrera: Agregador de datos para IoT nativo en la nube

Autor: Jose Fuentes Castillo
Tutor: Clara Isabel Luján Martínez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

El Secretario del Tribunal

0. Índice

1. Introducción

1.1. Motivación

Cada día aparecen nuevos productos domésticos con conexión a internet. Estos dispositivos se conectan a servicios online con los que interactúan, normalmente enviando datos. Estas flotas de millones de dispositivos enviando pequeñas ráfagas de datos son un perfil que hasta ahora no era común en internet. Esto puede suponer un reto para la forma tradicional de diseñar estos servicios.

El *cloud computing* es ya, *de facto*, el método elegido para ejecutar cargas de trabajo de servicios, pero es un concepto muy amplio y pocas veces bien explicado. El *cloud* ha evolucionado bastante desde sus inicios, y existen tendencias claras en la evolución del mismo que van a forzar la forma de funcionamiento de las aplicaciones de servidor. Las nuevas generaciones de *cloud* tienden a paradigmas en los que los recursos son totalmente dinámicos. Para que las aplicaciones funcionen bien sobre el *cloud* y aprovechen todo su potencial, se deben fijar ciertos patrones de diseño que hasta ahora no eran comunes.

Este proyecto pretende explorar cómo debería construirse en un sistema colector de datos para una aplicación de *Internet de la Cosas (IoT)* con un enfoque nativo en el *cloud*.

1.2. Objetivos

El objetivo del proyecto es aprender los requisitos de una plataforma que reciba y procese datos que podrían venir de una flota de dispositivos *IoT*, así como las limitaciones adicionales que añade implementar la solución con elementos nativos en el *cloud*.

A partir de ahí, se pretende desarrollar una base de este sistema, diseñando e implementando una solución que pueda asumir grandes ráfagas de datos y procesarlos en el *cloud*, ofreciendo una pila de protocolos conveniente para el mundo *IoT*.

Se espera producir una primera versión totalmente funcional, además de crear una base sólida que defina buenos patrones de desarrollo, y que pueda retomarse para solucionar casos de uso reales.

1.3. Estructura del documento

Este documento comienza describiendo la motivación y objetivos del trabajo realizado para a continuación, hacer un estudio sobre el estado del arte de los dos grandes campos que abarca el proyecto: *IoT* y *Cloud Computing* y que se recoge en el capítulo 2.

En el capítulo 3, se establecen los requisitos y define el alcance de la solución a implementar incluyendo un diseño pormenorizado de todos los subsistemas que conforman la solución.

A continuación, en el capítulo 4, se detallan las decisiones tomadas para la implementación. Se incorpora también como anexo el código fuente. En el capítulo 5 se muestran los resultados de esta implementación.

Por último, se incluyen una reflexión con las conclusiones que se desprenden de haber realizado este trabajo (capítulo 6), así como una serie de propuestas para futuros trabajos relacionados (capítulo 7).

2. Estado del arte

2.1. *IoT*

2.1.1. Origen de IoT

El primer uso del término *Internet of Things* se atribuye a Kevin Ashton, cofundador de *Auto-ID center*. *Auto-ID center* es un grupo de investigación centrado en el campo de RFID (*Radio-Frequency IDentification*). Según sus propios fundadores, la misión del grupo era investigar y desarrollar la tecnología para lo que vendría después del código de barras.

Parte de su ideario era que cada objeto estaría identificado electrónicamente de forma que, por ejemplo, un plato de spaghetti precocinado ofrecería información al microondas de forma que este supiera cómo debía calentarlo.

Este concepto parecía poder extenderse a cada objeto en el mundo, pudiendo ser estos localizados y potencialmente controlados gracias a esta identificación electrónica. Para hacer esto de forma eficiente, sabían que debían hacer los dispositivos lo más pequeños posibles (el silicio es caro). Una

de las formas en que pensaron para poder hacer estos dispositivos más económicos fue mover la memoria a otro sitio. Internet era el sitio obvio. Ahí se acuñó el término *Internet of Things*.

Como se adelantaba, el término fue acuñado por Kevin Ashton en 1999. Lo usó durante una demostración que estaba haciendo a la multinacional *P&G* (*Procter & Gamble Co*) en la que explicaba que, en realidad, estaban haciendo una extensión de internet para acomodarla a "las cosas". De ahí evolucionó el término *Internet of Things*[1]. En *AutoID* se empezó a usar en todas las demos y charlas y también se extendió fuera de esa institución.

Cabe mencionar también que, aunque parece que ese fue el primer uso del término, no fue la primera vez que alguien conectó "una cosa" a internet. Existen referencias a proyectos previos (muchos de ellos anecdóticos), como el caso de la "cafetera conectada" del *Cambridge Computer Lab* en 1993[2]. El concepto de webcam no existía por aquel entonces. La gente de este laboratorio conectó una pequeña cámara apuntando a la cafetera a la red interna. De esta forma los investigadores podía ver si había café disponible antes de dar el largo paseo hasta la habitación donde estaba la cafetera.

Internet se ha ido transformando poco a poco. En las últimas décadas, el uso que hacemos de internet ha ido mutando sustancialmente. Uno de estos cambios ha sido que las comunicaciones en internet ya no tienen porqué tener como origen y/o destino un ente humano. Existen dispositivos de todo tipo (*things*) que producen o consumen información que viaja a través de internet con destino otras máquinas. A esto de llama comunicación m2m (*machine to machine*).

2.1.2. Nuevas problemáticas que presenta IoT y nuevas tecnologías para resolverlas

El hecho de que potencialmente cualquier cosa se pueda conectar a internet crea multitud de retos.

2.1.2.a. Protocolos de servicio, transporte y red no óptimos

Internet no fue pensado dar cabida al paradigma del *IoT*. En primer lugar, aparece el problema del direccionamiento. A nivel de red, el direccionamiento recae en el protocolo *IPv4*. O al menos esto era así hasta hace poco tiempo. El protocolo *IPv4* usa direcciones de 32 bits, quedando limitado el número de las mismas a 2^{32} (4.294.967.296). Puede parecer un número muy grande, pero desde

[1] https://www.redbite.com/the-origin-of-the-internet-of-things/
[2] http://www.bbc.com/news/technology-20439301

luego no es suficiente para direccionar cada uno de los objetos que potencialmente podrían ser conectados.

IoT ha sido uno de los motivos de que se el direccionamiento en internet esté migrando a *IPv6*. Esta versión ofrece 2^{128} (~$340x10^{36}$) direcciones. Esta cantidad es equivalente a 670 mil billones de direcciones por cada milímetro cuadrado de la superficie de la tierra. Es de esperar que tal cantidad de direcciones sea suficiente para cualquiera de los usos previstos en *IoT*.

Otro inconveniente es que los protocolos de comunicación de más alto nivel tampoco no suelen estar optimizados para el caso de uso de pequeños dispositivos, normalmente a batería, con poca capacidad de cómputo, poco ancho de banda y conectividad intermitente. Imagínese un termómetro con capacidades de conexión a internet. Dicho dispositivo enviaría cada minuto la temperatura medida. El dato en sí de la temperatura es insignificantemente pequeño, del orden de varios bytes a lo sumo. Sin embargo, la cantidad de datos de señalización introducidos por los protocolos inferiores (*HTTP*, *TCP*, etc) sería como mínimo uno o varios órdenes de magnitud mayor.

Para resolver este problema surgió el protocolo *CoAP* (*Constrained Application Protocol*). Se podría decir que este protocolo es el equivalente a *HTTP* pero sobre *UDP*. Frente a *HTTP*, tiene la ventaja de que es mucho más ligero (introduce mucha menos información de señalización) y que además soporta *multicast*.

2.1.2.b. Seguridad

El hecho de conectar un dispositivo a internet implica exponerlo a inimaginables fuentes de ataques. Normalmente, los ordenadores personales, servidores y equipos industriales que tienen conexión a internet suelen mantener su software actualizado, y es muy frecuente aplicar parches de seguridad. En dispositivos de *IoT* no es tan sencillo. Para empezar, muchos de ellos son equipos desatendidos a los que se les realiza poco o ningún mantenimiento. Y además, suelen disponer de un software bastante sencillo y pocas capacidades para realizar operaciones como una actualización completa del firmware por ellos mismos.

Con el tiempo se han ido incorporando soluciones que permiten actualizar el firmware de una flota de dispositivos IoT de forma fácil para el usuario. Ahora es habitual que el smartphone haga las veces de una pasarela que permite a dispositivos del tipo bombillas inteligentes actualizar su firmware.

En otras soluciones *IoT* más industriales, existen otro tipo de pasarelas (*gateways*) dedicados, que tienen un inventario de dispositivos conectados y pueden incluso integrarse con sus *bootloaders* para instalar una nueva versión de su software.

2.1.2.c. Procesamiento de grandes cantidades de datos

Una de las aplicaciones más extendidas en *IoT* es *data gathering*. Consiste básicamente en recolección de multitud de datos de muchos dispositivos distribuidos.

Recolectar y almacenar esta información en bruto no es un gran problema. La problemática real es como indexar, organizar y correlar toda esa información.

El manejo de estos grandes *datasets* ha obligado al software del lado servidor a evolucionar. Se ha pasado de las tradicionales bases de datos *SQL*, en las que los datos están ordenados en tablas y columnas sobre las que se pueden realizar multitud de operaciones (buscar, ordenar según cualquier campo, unir...) a sistemas de clave-valor. Estos sistemas son mucho más simples. Simplemente permiten guardar un dato y referenciar con una clave luego puede usarse para volver a obtenerlo. El software que hace uso de estos datos tiene que cambiar la forma de trabajar. Son necesarios nuevos algoritmos para organizar la información. Cobra muchas más importancia el procesado previo al almacenamiento.

Las aplicaciones de *data gathering*, uno de los grandes campos dentro de *IoT*, presentan otro problema relacionado. Dichas aplicaciones ingestan (miles de) millones de eventos. Estos eventos tienen cierta *metadata* asociada (magnitud medida, dispositivo de captura, condiciones alrededor del evento, usuario, etc) y además tienen una huella temporal. Es común que con ese cantidad masiva de eventos se quieran hacer operaciones del tipo correlación temporal entre algunos de los parámetros, o simplemente conteos de eventos que se filtran de acuerdo a criterios en la *metadata*. Una base de datos de tipo *SQL* es bastante ineficiente para este tipo de trabajo. Así que en los últimos años se han desarrollado multitud de tecnologías de bases de datos de eventos o bases de datos orientadas a series temporales. Dos de las más usadas son *InfluxDB*[3] y *Elasticsearch*[4].

[3] https://github.com/influxdata/influxdb
[4] https://github.com/elastic/elasticsearch

2.1.3. Aplicaciones domésticas de *IoT*

El público general tiene ya a su disposición multitud de aplicaciones que podrían clasificarse como soluciones *IoT*.

A continuación se intenta dar una visión general de las aplicaciones existentes en este campo. El espectro de diferentes aplicaciones es inmenso y es imposible cubrirlo entero en este documento, pero se ha intentado escoger algunos de los ejemplos más representativos o interesantes.

2.1.3.a. *Smart Lighting*

Existen multitud de soluciones comerciales de *Smart Lighting*. Este tipo de dispositivos se componen de bombillas con cierta inteligencia y conectividad, mandos remotos, y un *gateway* que los conecta a internet. Las bombillas, mandos y *gateway* están interconectados mediante una red tipo *WLAN*[5] basada en *IEEE 802.11* (Wi-Fi) o mediante una red *WPAN*[6] basada en *IEEE 802.15* (*Bluetooth, ZigBee* u otras implementaciones propietarias). El *gateway* se conecta a la red local doméstica, por la que accede a internet. Eso abre la puerta a multitud de servicios, como control remoto desde el smartphone o desde una web.

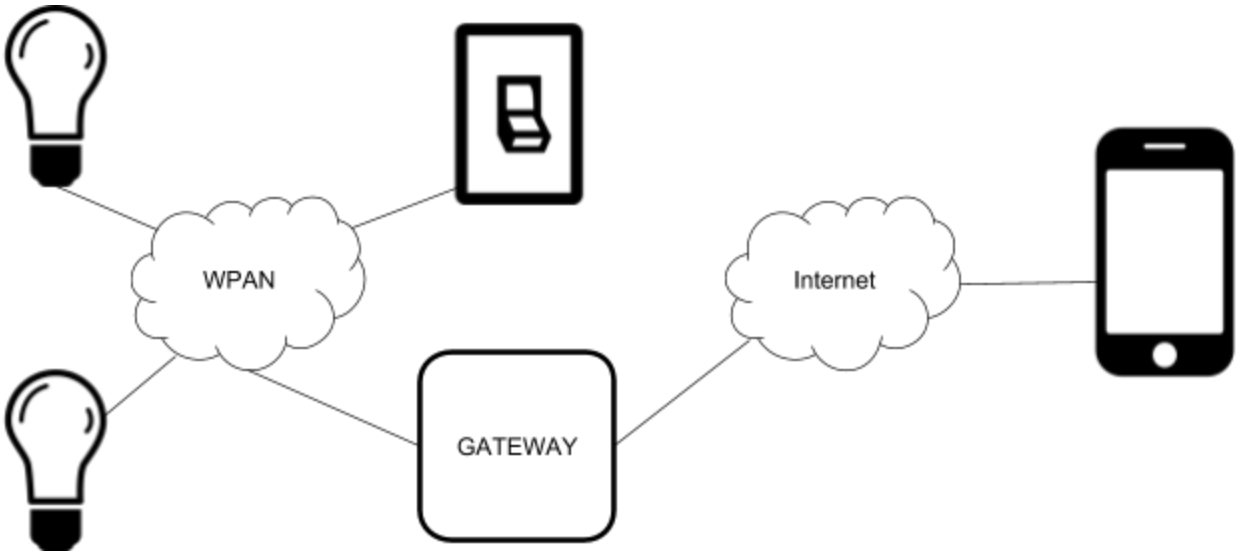


Fig 1. Esquema de conectividad de sistema *Smart Lighting*.

[5] Wireless *Local Access Network*
[6] Wireless Personal Area Network

Ejemplos de soluciones comerciales de este tipo son *Philips Hue*[7] o *Ikea TRÅDFRI*[8].

2.1.3.b. *Smart Home Solutions*

Se podría diferenciar otro grupo de soluciones que incorporan una arquitectura similar a la anterior, pero con otros sensores y actuadores. Incluso luces. La arquitectura sería similar a la anterior: diferentes "periféricos" conectados a una *WLAN* o *WPAN*, o ambas, envían datos y/o reciben instrucciones a través de un *gateway*.

Como ejemplos de periféricos se tendría: sensores de presencia, cámaras, sensores de puertas y ventanas, sensores de temperatura, sensores de humos, enchufes que miden el consumo, enchufes que pueden ser actuados remotamente, altavoces, micrófonos, mirillas electrónicas, bombillas o paneles led e incluso aspiradoras.

Por otro lado, gracias a que todo el sistema tiene conectividad con internet, es posible la existencia de servicios que permiten definir reglas que interconectan todos estos dispositivos, incluso entre diferentes hogares. Es posible, por ejemplo, hacer que cuando no haya nadie en la casa (situación detectada por los sensores de presencia o incluso por las cámaras), se apaguen todas las luces y se ponga la aspiradora robot en modo patrulla. También sería posible abrir la puerta al mensajero e incluso interactuar con él desde el móvil, las cámaras, micrófono y altavoz de la la casa.

Gracias a los enchufes controlables, es posible conectar dispositivos que, a priori, no están diseñados para estar conectados, como por ejemplo un simple ventilador.

También existen dispositivos que permiten integrar otros aparatos no inteligentes que tienen mando a distancia. Estos dispositivos vienen a ser un mando a distancia universal, que además está conectado a internet. Hacer posible, por ejemplo, controlar un aire acondicionado o la televisión desde el smartphone, desde cualquier sitio.

Existen otros dispositivos que se conectan a las placas de control de aires acondicionados o calefacciones y hacen de *gateway* con internet.

[7] https://www2.meethue.com/en-us
[8] http://www.ikea.com/gb/en/products/lighting/smart-lighting/

Ejemplos populares de estos sistemas *smart home* serían *Xiaomi Mi Smart Home*[9], *Logitech Harmony* [10] o *Nest*[11].

2.1.3.c. Pulseras actividad

Los últimos cinco años han sido el boom de los *activity trackers*. Estos dispositivos forman parte de los llamados *wearables*, dispositivos que las personas llevan puestos. Cuentan con diferentes sensores que miden diferentes magnitudes (pasos, saltos, movimiento, pulso, oxígeno en sangre, respiración, etc.). Usos datos sufren un primer procesamiento en el dispositivo para ser enviados, por internet, hasta la plataforma online que ofrece el servicio. Este servicio procesa los datos para cuantificar información que tienen cierto interés para el usuario.

Con estos productos es posible cuantificar aspectos como el sueño o la actividad diaria. Algunos servicios permiten al usuario introducir datos adicionales como la comida ingerida, y ofrecen planes de entrenamiento y dieta, de los cuales el usuario puede hacer un seguimiento él mismo a través de una aplicación.

Ejemplos de estos dispositivos son *Fitbit Flex*[12], *Oura Ring*[13] o *Polar Loop*[14].

2.1.3.d. *Dog trackers*

Gracias al desarrollo de la telefónica *m2m*[15] (acceso a la red telefónica e internet para máquinas) es posible tener "flotas" de muchos dispositivos en las que cada uno tiene su propio acceso a internet (o a una red *WAN*[16] privada).

Si a esto se le añade la tecnología *GPS*, es posible construir un dispositivo pequeño que se puede colocar en el collar de un perro y permite localizarlo a través de un smartphone en todo momento.

Un ejemplo de este servicio es *Tractive*[17].

---

[9] https://www.androidcentral.com/xiaomi-mi-smart-home-kit-home-automation
[10] https://www.myharmony.com/en-es/
[11] https://nest.com/
[12] https://www.fitbit.com/es/flex2
[13] https://ouraring.com/
[14] https://www.polar.com/en/products/fitness_crosstraining/loop2
[15] Machine to machine
[16] Wide Access Network
[17] https://tractive.com/eu_es

2.1.4. Aplicaciones industriales de *IoT*

2.1.4.a. *Fleet tracking*

En la sección anterior se ha hablado sobre los *dog trackers*. La tecnología detrás de las soluciones de *fleet tracking* (localización de flotas) es en parte la misma: accesos de datos *m2m* y *GPS*. En este caso se podría añadir una integración extra, y es que los vehículos modernos tienen una interfaz que permite interactuar con diferentes sistemas del mismo. Algunas soluciones de fleet tracking permiten parar remotamente un vehículo en caso de robo (o automáticamente si, por ejemplo, sale de una zona geográfica determinada).

Por ejemplo, la compañía *muving* de alquiler de scooters eléctricos controla su flota con un sistema de este tipo. En todo momento saben dónde está cada motocicleta y la batería que tiene disponible. El usuario desde su smartphone, a través de los servidores de *muving*, ve donde está el vehículo, arranca o para la moto, y abre el baúl donde se guarda el casco.

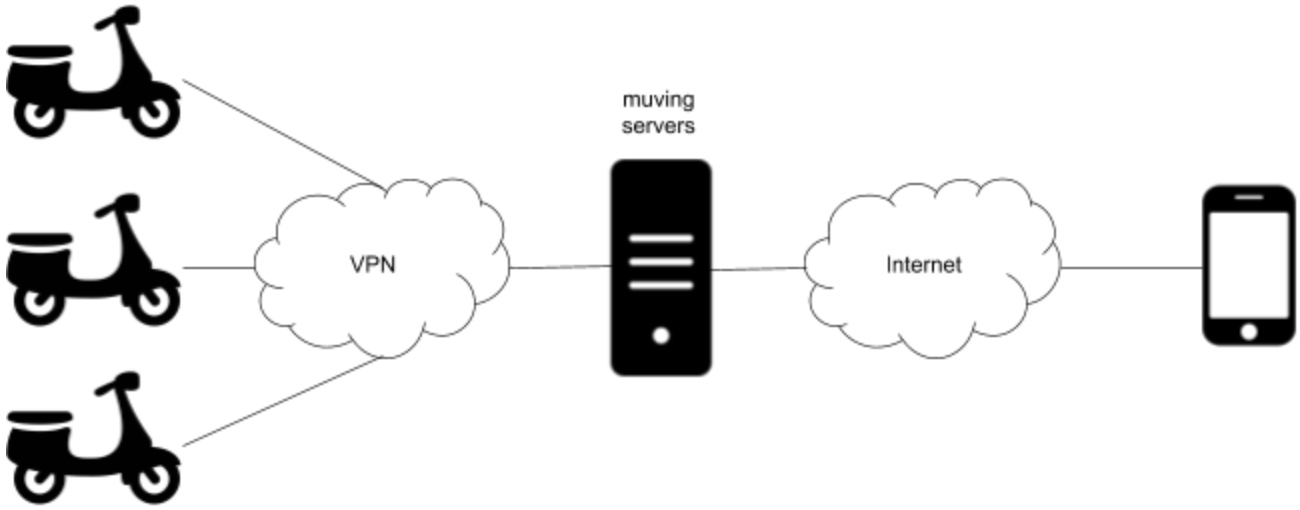


Fig 2. Esquema de conectividad de flota de *muving*.

2.1.4.b. Monitorización de tráfico

Los centros de control de tráfico son sistemas de *IoT* en cierto modo. Reciben información de millones de cámaras y sensores remotos. Quizás siendo estrictos no podríamos catalogarlo como *IoT*

puesto que esos datos no salen públicamente a Internet, sino que viajan a través de túneles privados por internet hasta su destino.

Sin embargo hay un caso de monitorización de tráfico que encaja perfectamente en la definición de *IoT*: *Google Traffic*. *Google Traffic* es una característica de *Google Maps* que ofrece información del tráfico en tiempo real. Esa información es usada por el navegador para calcular la ruta más rápida y evitar atascos.

Para que el sistema funcione, es necesario recolectar millones de datos de ubicación de usuarios. A este tipo de prácticas se las llama *crowdsourcing*. No es descabellado afirmar que en cada coche viaja como mínimo un smartphone. El smartphone conoce su ubicación geográfica (bien por *GPS*, o por triangulación de red). Esta información es reportada a los servidores de *Google*. El sistema digiere toda esa información, aplica algoritmos predictivos y ofrece a los usuarios de los mapas de *Google* una visión de cómo está el tráfico en su ruta y cuál es el camino más conveniente para llegar a su destino.

2.1.4.c. Mejora de servicios en la ciudad gracias a *IoT*

En los últimos años también han surgido algunas aplicaciones de *IoT* destinadas a mejorar como funcionan los servicios en una ciudad, dando lugar al nacimiento del término *Smart Cities*.

Uno de estas aplicaciones es la telelectura de contadores. Los contadores de suministro (agua, gas o electricidad) están evolucionando para incorporar la capacidad de enviar sus lectura telemáticamente. Una de las formas de implementar esta funcionalidad para por usar acceso telefónico *m2m*.

Otra aplicación relacionada es la del alumbrado público telecontrolado. Es posible embeber dispositivos de control remotos en las farolas que iluminan la calle de forma que son controladas punto a punto vía internet. La ventaja en este caso es que se puede centralizar el control del alumbrado en un sólo sistema inteligente que ajusta la iluminación según las condiciones.

También se está empezando a añadir sensores a los cubos de basura. Esto permite a un sistema planificar de forma inteligente la ruta de recogida de basura, aumentando la eficiencia, reduciendo costes y evitando que haya basura amontonada sin que nadie vaya a recogerla por desconocimiento de las personas competentes.

2.1.5. Aplicaciones experimentales de *IoT*

Realmente no se ha hecho más que mostrar algunos ejemplos significativos de cómo se está usando *IoT* en la vida real. No hay duda de que esto es una realidad y de que con el tiempo irán apareciendo más y más complejas aplicaciones.

Basta hacer una búsqueda rápida en internet para encontrar ejemplos de gente de a pie haciendo sus propias integraciones entre servicios y *gadgets*.

He aquí un ejemplo curioso. Otro de los campos que está evolucionando a pasos agigantados es la inteligencia artificial. Tanto es así que ya hay soluciones comerciales bastante válidas que permiten, por ejemplo, dar órdenes a un ordenador manteniendo una conversación natural con el.

Conectando un servicio como el descrito con alguno de las aplicaciones de *IoT* vistas anteriormente, se pueden hacer cosas interesantes como controlar la iluminación de una casa vía voz. Esto es totalmente posible mediante la conjunción de *Amazon Alexa* y *Philips Hue*[18].

Es muy interesante el hecho de que muchos de los sistemas aquí citados ofrecen *APIs* documentadas para interactuar con sus sistemas, de forma que un usuario con conocimientos de programación podría fácilmente interconectarlos entre ellos, o incorporar elementos como el mencionado *Alexa*, u otros equivalentes.

2.2. *Cloud technologies*

2.2.1. Origen del cloud computing

Tradicionalmente cualquier servicio informático era servido desde desde una computadora. Una computadora tiene tiene recursos hardware sobre los que se ejecuta una pila de software a diferentes niveles. Desde el sistema operativo, que tiene interacción directa con el hardware, hasta cualquier aplicación en el espacio de usuario.

2.2.1.a. El término "*cloud*"

El origen del término *cloud* no es más que marketing y, desafortunadamente, da a pensar que se trata de algo místico y casi mágico. En realidad, debería permanecer solamente una de las

[18] https://www2.meethue.com/en-us/friends-of-hue/amazon-alexa

cualidades de las nubes cuando aplicamos este término. "*El cloud*" es algo efímero, al igual que las nubes en el cielo. Nada en "*el cloud*" se puede asumir como permanente. "*El cloud*" puede y debe verse como una pila de recursos (no infinitos, pero sí significativamente grandes) de la que se puede tomar prestado temporalmente lo que se necesite para un cierto cometido, hacer uso de ello durante un tiempo, y liberarlo más tarde. En este contexto, con recursos se entiende una capacidad de cómputo, de almacenamiento o de red, que se maneja en forma de abstracciones que más adelante se cubren en este texto.

Es común que exista un "proveedor de *cloud*" que ofrece estos recursos como servicio, a cambio de una cierta suma de dinero.

Existen multitud de proveedores de este tipo de servicios. El pionero fue *Amazon Web Services (AWS)*, pero le siguen (muy) de cerca otros proveedores como *Google Cloud* o *Microsoft Azure*.

2.2.1.b. El origen

El nacimiento de lo que hoy llamamos "*cloud*" puede datarse en 2003 en una exitosa tienda online, *Amazon*. El tamaño de *Amazon* era tal que para su correcto funcionamiento requería un compleja infraestructura informática. De algunos de los ingenieros encargados de diseñar, implementar y mantener dicha infraestructura surgieron grandes ideas para cambiar la forma en la que un sistema de ese tipo funcionaba.

Ya existían las máquinas virtuales (de las que más adelante se hablará), pero ellos idearon *EC2*. *EC2* es un servidor de máquinas virtuales, una *API* que permite crear, gestionar, dimensionar y hacer todo tipo de operaciones con *VMs*. También desarrollaron mucha tecnología alrededor de *EC2*. Crearon servicios de volúmenes de datos y toda una capa de networking especialmente diseñada para este dinámico sistema. Es especialmente interesante esta capa de red, puesto que revuelve muchos problemas clásicos en las redes de una forma totalmente distribuida: e.g. en *AWS* existe un concepto de firewall distribuido en forma de *Security Groups*, que controla qué comunicaciones de red son posibles entre los elementos del cloud.

Cuando empezaron a desarrollar esas ideas como soluciones a problemas internos, surgió en paralelo la idea de comercializarlo. *AWS* fue lanzado en 2006 y resultó ser el origen de un negocio con beneficios anuales mayores a 12.000 millones de dólares (2016) y con un crecimiento que parece no tener fin.

2.2.2. Problemas que el cloud computing intenta resolver

2.2.2.a. El problema del escalado

Digamos que se quiere ofrecer un servicio de email, se deberá preparar una máquina con capacidad suficiente para dar ese servicio e instalar sobre ella el software adecuado. ¿Qué sucede si tras un tiempo crece el número de usuarios o la intensidad con que estos usuarios usan el servicio?. Puede que el hardware no sea lo suficiente potente para satisfacer las necesidades que el software demanda para esa cantidad de usuarios. En ese caso habría escalar el servicio.

No existe una fórmula única para escalar un servicio. La forma en que una aplicación escala, está fijada por diseño de la misma. Hay aplicaciones que sólo pueden ser escaladas aumentando los recursos de los que dispone, y esa relación puede ser lineal o no. Es muy común que sea de un orden mayor, lo cual haga inviable su operación llegada una cierta demanda.

Sin embargo, otras aplicaciones están pensadas para poder ejecutarse en topologías de alta disponibilidad (comúnmente *HA*, *High Availability*). Una de las topologías *HA* más habituales es repetir varias instancias de la aplicación en sí, y balancear la carga entre las mismas poniendo un elemento delante de estas réplicas.

Esto resulta conveniente cuando los recursos consumidos por una aplicación según el número de clientes que la usan no es lineal, sino que tiene un orden mayor que uno.

Se aprecia aquí una situación paradójica. Si se pudieran dividir los recursos de una misma máquina entre dos instancias del software, podríamos dar servicio a más usuarios.

El problema de ejecutar más de una instancia de un servicio sobre el mismo hardware no es algo trivial, como se verá en el siguiente apartado.

2.2.2.b. El problema de compartir recursos: virtualización

Sea una máquina con ciertos recursos disponibles que se quieren compartir por más de una aplicación o servicio, de naturaleza muy diferente e incluso de propietarios diferentes.

Esto plantea muchos problemas y levanta muchas preocupaciones.

Para empezar habría que garantizar una cierta separación entre estas aplicaciones. Habría que garantizar que la información perteneciente a los usuarios del servicio A no es accesible bajo ningún concepto por el servicio B (ya tenga el servicio B la intención de acceder a esa información o no).

También habría que establecer unas políticas que permitan distribuir adecuadamente los recursos disponibles y evitar que los abusos de una aplicación perjudique a otra.

Se suma la problemática de que hay algunos recursos que no se pueden aumentar añadiendo hardware más potente. Es el caso de recursos lógicos como los puertos. Si un servicio ocupa el puerto 80 para dar un servicio web, no podrá haber otro servicio en la misma máquina ocupando ese puerto.

Se ve que el problema que planteamos reside en que tradicionalmente el software y el hardware están íntimamente acoplados. El hecho de repartir unidades de recursos predeterminadas y físicamente definidas (máquinas) para instancias de software (o viceversa), limita enormemente la forma en que se pueden aprovechar los recursos.

Este problema se lleva tiempo resolviendo a base de *virtualización*.

La virtualización consiste en poner una capa de software (un hipervisor) sobre el hardware, de forma que sobre el hipervisor se puede ejecutar más de un sistema operativo. Estos sistemas operativos tienen visión exclusiva del hardware, y están aislados del sistema operativo vecino. El hipervisor hace de interfaz entre el hardware real y el sistema operativo, presentando a este unos recursos hardware que en realidad son abstracciones virtuales.

2.2.2.b.a. Virtualización y paravirtualización

Para evitar confusiones, se hará referencia a estos términos como *full-virtualization* y *paravirtualization*.

En el caso de full virtualization, el sistema operativo virtualizado, *guest*, no es consciente de que no se está ejecutando sobre un hardware real. Esto implica que sus drivers hablarán con el hardware virtualizado como si fuera hardware real, con los mismos sets de instrucciones, con comandos propios del hardware. Como consecuencia, el sistema operativo host, que contiene el hipervisor, debe hacer una traducción de instrucciones, comúnmente llamada recompilación dinámica o *dynamic recompilation*. Esta interfaz doble tiene un coste en el rendimiento.

En el caso de *paravirtualization*, el sistema operativo *guest* es consciente de que está en un entorno virtual. Tiene unos drivers específicos que, en lugar de usar comandos hardware, emiten comandos dirigidos directamente al sistema operativo *host*. Esto evita gran parte del trabajo de traducción de instrucciones, por lo que el rendimiento en general mejora.

2.2.2.b.b. Virtualización asistida por hardware

Existen microprocesadores que facilitan al hipervisor la tarea de traducir instrucciones. Estos procesadores disponen de instrucciones para la creación de un contexto en el que las instrucciones del guest pueden ser ejecutadas de forma segura. El *guest* podría incluso ejecutar instrucciones con privilegios sin afectar al *host*.

La mayoría de procesadores modernos tienen sets de instrucciones dedicados a estos menesteres. Esto hace que la técnica *full-virtualization* esté muy extendida en la industria.

El cloud hace uso de la virtualización para resolver el problema de la compartición de recursos.

2.2.2.c. El problema de los monstruos monolíticos: microservicios

Tradicionalmente, es normal encontrar grandes y complejos servicios que se sirven desde una única aplicación monolítica. En estos, todo se ejecuta en una misma máquina que centraliza todo el trabajo.

Este tipo de arquitectura limita mucho la posibilidades de evolución del servicio, ya todas las tecnologías deben convivir sin conflictos, las interfaces entre funcionalidades se hacen totalmente difusas y todo se vuelve especialmente complejo en asuntos claves como la escalabilidad.

El paradigma de los microservicios propone crear unidades separadas que cubran pequeñas necesidades. Estas unidades son fáciles de entender, testear, desarrollar, mantener y escalar por separado.

El cloud está pensado para ser aprovechado con microservicios.

2.2.2.d. El problema de la heterogeneidad

Para componer un servicio complejo, es común que haya que usar tecnologías muy diferentes entre sí.

Cada aplicación se instala, configura y ejecuta de forma diferente. En particular, los diferentes componentes de un servicio tienen diferentes requisitos de *runtime*: sistema operativo, paquetes del sistema, librerías adicionales, etc. En ocasiones, esos requisitos de runtime pueden llegar a ser incompatibles entre sí.

Esto suele implicar preparar el sistema operativo de forma minuciosa para que todo funcione como una máquina bien engrasada: crear usuarios, dar los permisos adecuados, instalar las dependencias que satisfagan las necesidades de todos los componentes, evitar incompatibilidades, gestionar logs, gestionar cómo arrancan los servicios, gestionar donde escriben y de donde leen, controlar los recursos que consumen, y esta lista podría continuar.

No hay una solución única para este problema en el *cloud*, pero una buena estrategia pasa por hacer microservicios (hacer que los runtimes de cada microservicio sean independientes).

2.2.3. Problemas que el *cloud computing* crea

Para usar eficientemente el *cloud*, un sistema deberá funcionar bajo ciertas suposiciones.

2.2.3.a. Costes dinámicos

Puesto que la reserva de recursos es dinámica, el ciclo de vida de una máquina (virtual) en el *cloud* es clave. Cuando una máquina se apaga, sus recursos son liberados (y por tanto dejamos de pagar por ellos), esa máquina deja de existir. Existen mecanismos para asegurar que se pueden mantener los datos y que podemos restaurar el estado. Algo similar ocurre con otros elementos de la infraestructura: *networks*, *load balancers*, *firewalls*, *data volumes*, etc.

Una de las promesas del *cloud computing* es eficiencia y ahorro de costes. No es en absoluto trivial ni inmediato hacer que una solución en el cloud cumpla esa promesa.

Supóngase el caso en que se tiene un servicio corriendo actualmente en un data center tradicional. Ese servicio tiene unos recursos asociados que son estáticos y dedicados. Una migración al cloud directa de ese servicio reservaría los mismos recursos de forma estática. Eso sería el "camino fácil", pero el coste es alto: el precio de los recursos en el cloud se suele facturar por tiempo (minutos, segundos u horas) y si se hace el cálculo suponiendo que esos recursos están reservados 24/7 los costes se disparan.

Un uso eficiente del *cloud* sería sólo reservar recursos cuando de verdad se necesiten. El *cloud* permite, por ejemplo, redimensionar las máquinas de forma fácil (incluso sin apagarlas).

Por tanto, alrededor de una aplicación tradicional, habría que añadir ciertos elementos integrar capacidades como escalado de nodos, o almacenamientos en volúmenes de datos nativos.

También es posible (y seguramente más rentable) resolver el mismo problema que el servicio original pero con elementos nativos del cloud. Por ejemplo, un servicio de almacenamiento masivo de datos en una migración directa al cloud terminaría siendo una máquina corriendo 24/7 con un disco bastante grande más lo necesario para asegurar la integridad de los datos (posiblemente discos adicionales). Una implementación nativa de la solución en *AWS* sería usar una *función lambda*[19] que guarda datos en *S3*[20]. Se ahondará en estos términos más adelante.

2.2.3.b. Falta de control de la disponibilidad de los recursos

En *cloud computing*, las cargas de trabajo se reparten pseudo-arbitrariamente sobre los *pools* de recursos. Esto significa que en una misma máquina física se están ejecutando procesos de diferentes usuarios. Si un usuario bloquea la CPU, puede llegar a perjudicar a otro. Normalmente los *cloud providers* lidian con estos problemas con penalizaciones en ciclos a los usuarios que causan bloqueos (limitan el uso de procesador ese esos procesos por un tiempo).

Otro fenómeno interesante es que puede darse el caso de que usuarios tengan reservados gran cantidad de recursos que realmente no estén usando. En este caso los proveedores suelen vender el uso de esos recursos a otros usuarios a un menor coste, a condición de que en cualquier momento pueden ser "desalojados" para devolverlos a sus arrendatarios originales.

Los proveedores de cloud ofrecen diferentes garantías de nivel de servicio. Evidentemente, a mayor disponibilidad garantizada, mayor precio.

Una solución avanzada que lidiara con problemas de disponibilidad de recursos a nivel de aplicación permitiría disminuir mucho los costes.

---

[19] *AWS Lambda* permite definir funciones y servirlas en base a eventos (por ejemplo peticiones HTTP). AWS sólamente facturará el coste de procesamiento de la función cuando es ejecutada. Más información en aws.amazon.com/lambda.

[20] *AWS S3* es un servicio de almacenamiento de datos. AWS asegura la integridad de los mismos de forma transparente al usuario (no hay necesidad de preocuparse por hacer copias de seguridad). Los precios de *S3* dependen de cuánto se acceda a los datos y del tamaño de los mismos.

En cualquier caso y al igual que pasaría en un data center tradicional, el *SLA*[21] nunca es 100% y aunque las máquinas físicas y sus problemas están abstraídos para los usuarios del cloud, puede ocurrir que una avería física en las máquinas reales cause *downtime* en el servicio. La diferencia con un *data center* tradicional es que en esos casos no hay nada en manos del usuario que pueda hacerse para acelerar el arreglo de la situación. Esto no es necesariamente un problema, ya que, si el proveedor es bueno, seguramente el tiempo de reemplazamiento de la carga del usuario en otra máquina es inferior de lo que se tardaría en solventar la situación en un *data center* tradicional.

2.2.3.c. Hardware limitado

Para ciertas aplicaciones se requiere un hardware muy específico. Normalmente los proveedores de *cloud* ofrecen muchas y variadas opciones: instancias con almacenamiento ultrarrápido, instancias optimizadas para procesamientos gráficos, instancias con muchos núcleos, etc. Sin embargo en ningún caso se tendrá control absoluto sobre el hardware que hay debajo. Si el software a migrar es íntimamente dependiente del hardware, puede que el cloud no sea una opción.

2.2.3.d. *Provider lock-in*

Una de las cosas que al principio causan rechazo a desarrollar soluciones *cloud native* es el llamado *lock-in*.

Si se hace una gran inversión en adoptar diferentes tecnologías privativas ofrecidas por un proveedor de cloud, termina siendo difícil moverse a otro proveedor, puesto que el coste de migrar todo de nuevo sería muy alto.

Frente a esto no hay una única solución, pero sí diferentes estrategias como usar diseños muy modulares que permitan conmutar partes de la solución manteniendo las interfaces. Por ejemplo: usar una API de almacenamiento agnóstica como *minio*[22] que permita conectarse tanto a *AWS S3*, como a *Google Cloud Storage* permitiría facilitar una migración de proveedor sin grandes dramas.

[21] *Service Level Agreement*
[22] Minio (github.com/minio/minio) ofrece una API para el almacenamiento de archivos que es compatible con S3 y otros backends similares.

2.2.3.e. Viabilidad de una aplicación en el cloud

Todo en el cloud ha de ser entendido como una descripción de recursos. Como el cloud provea esos recursos escapa a nuestro control. Cuando se piensa en ejecutar cierta cargar en el cloud, primero han de extraerse los requisitos de ella.

Es necesario conocer cómo es la arquitectura de la aplicación. Cada componente, cada relación, cada conexión va a tener su equivalente en el cloud, y no hay una solución única para realizar esa conversión. Para resolver este problema hay que identificar los componentes o procesos de la aplicación. A continuación es necesario identificar los requisitos de *runtime* de cada uno de ellos. Tras esto es conveniente evaluar si es posible y conveniente reemplazar alguno de ellos con elementos nativos del *cloud*[23]. Más allá de los componentes individuales, a nivel de red habría que pensar cuales son las conexiones entre los componentes y también cuales con las conexiones desde y hacia el exterior. Desde el punto de vista de la escalabilidad, habría que contemplar que componentes es posible escalar horizontalmente[24].

En un paso posterior habría que dimensionar la solución. Esto es equivalente a elegir la cantidad de CPU y RAM que, en media, consume cada componente. También habría que conocer el máximo permitido de dichas magnitudes. Estos datos permitirían elegir correctamente el tipo de instancia[25] necesario y el número de dichas instancias.

Es necesario conocer y monitorizar la dinámica del servicio para saber si todo debe estar corriendo todo el tiempo o es posible desconectar y liberar recursos en ciertos momentos. Es conveniente realizar una inversión en este aspecto, un sistema que se autoescala de forma programática es muy eficiente es coste.

En cualquier caso, la parte de dimensionamiento es menos importante que la parte de arquitectura, ya que la segunda es muy fácil de cambiar a posteriori. La primera, sin embargo, es condicionante desde el principio. En concreto, dar respuesta a cuáles son los requisitos de *runtime* de cada componente es muy importante por varios motivos. Tanto es así que se le dedica el siguiente apartado por completo.

---

[23] El "puedo" suele ser fácil de responder, el "debo" depende de muchos factores como: coste vs beneficio, *lock-in*, flexibilidad.
[24] Escalar horizontalmente es mantener más de una réplica de la instancia ejecutándose en paralelo. Esto no es trivial y requiere de que la aplicación en sí esté preparada. Es algo deseable en el cloud, puesto permite hacer diseños más robustos.
[25] Abstracción de computadora. Normalmente define el tipo de procesador, disco y otros componentes.

2.2.3.f. Requisitos de *runtime* de los componentes de una aplicación

En primer lugar habría que definir que es un *runtime*. Se considera *runtime* al entorno en el que se ejecuta el proceso. El *runtime* de un proceso debe tener disponibles las librerías necesarias por este para funcionar. Debe disponer también de un intérprete para el lenguaje en el que ese componente esté implementado. También debe tener cierta configuración que puede tener forma de archivos de configuración o de variables de entorno (por ejemplo). Dentro de la definición de *runtime* también se incluyen los directorios del sistema de archivos de donde el proceso va a leer o escribir, así como los puertos en los que va a poder escuchar.

Cuanto más exacta y precisa sea la descripción de esos aspectos, mejor se podrá configurar esta solución *cloud computing* y más efectiva será. Se verá que esto es más así con cada nueva generación de cloud (se cubrirá en los siguientes apartados).

En general, las aplicaciones que se adaptan bien al *cloud* son aplicaciones que mantienen todo su estado separado. Esto significa que se podría guardar el estado en un disco persistente, apagar la máquina, encender otra máquina, conectar el disco y volver a restaurar el estado, haciendo que la aplicación siga funcionando.

Este concepto es muy útil y permite hacer cosas muy interesantes que verán más adelante, como *rolling updates*[26]. Además, al ser todos los recursos efímeros, es importante poder reconstruir un entorno completo para la aplicación de forma fácil.

2.2.4. Tecnología madura de *cloud computing*: *Virtual Machines*

2.2.4.a. Fundamentos

Se ha hablado anteriormente de virtualización y se había fijado el origen de lo que se conoce como *cloud computing* en *AWS EC2*.

Una plataforma de *cloud computing* que ofrece *Virtual Machines* (abreviado *VMs*) consiste en lo siguiente:

[26] *Rolling update* es una técnica por la que se tienen N réplicas de un componente funcionando con una versión A, y progresivamente se van sustituyendo una por una por la versión B del componente, hasta que las N tienen la nueva versión. La ventaja es que se pueden hacer actualizaciones de la aplicación sin *downtime*.

El usuario interactúa con la plataforma a través de una *API* (generalmente se ofrecen portales web *point-and-click* que permiten consumir esa API de forma cómoda y gráfica). Esta *API* abstrae los recursos que se pueden crear y manejar en el *cloud* de forma que el usuario puede operar con ellos y hacer operaciones del tipo: *crear/modificar/eliminar: VM, volúmen de datos, redes, load-balancers, reglas de firewalls, etc.* Cada *cloud* tiene sus propias entidades, las cuales a veces son equivalente y otras muchas veces no.

2.2.4.b. Ciclo de vida de una *VM*

También es propio de cada *cloud* el ciclo de vida de las *VMs*. Es común que se ofrezcan varias imágenes base de las que partir para construir nuestras *VMs* propias. Se podría decir que una imagen base es una captura del disco de una máquina con el sistema operativo instalado. Hay imágenes más ligeras e imágenes más completas, pero es normal que para poder usarla haya que hacer personalizaciones propias. En ese caso, y según el *cloud*, se podría exportar una imagen propia para usarla como base en futuros despliegues. Es común que los *cloud* providers ofrezcan *hooks* para los distintos puntos del ciclo de vida de las *VMs*. Un *hook* es una entrada que el sistema ofrece para que un usuario personalice la lógica que se ejecutará en un cierto momento.

Los diferentes momentos del ciclo de vida cambian en cada *cloud* pero, en general, son bastante parecidos a los siguientes:

- *Build time*: periodo en que se crea una imagen base. Realmente está desligado del funcionamiento de la *VM*, es anterior a que realmente se esté ejecutando. Aquí se hacen cosas como instalar paquetes en el sistema.
- *First boot*: primera vez que arranca la máquina. Se suelen hacer cosas como configurar software con parámetros que solo se conocen en *runtime* (e.g. dirección, usuario y contraseña de la base de datos), inicializar la base de datos, o el volúmen de almacenamiento.
- *Boot:* todos los siguientes arranques de la máquina. Se suelen arrancar servicios.
- Restart: algunos *cloud providers* permiten hacer un reseteo ligero consistente en reiniciar el sistema operativo.
- *Halt*: parada de la máquina. Al volver a arrancar, segun *cloud provider*, puede que el disco haya vuelto a su estado inicial (los directorios que no estés persistidos en volúmenes).

- *Terminate*: borrado total de la máquina. Aún aquí, los datos persistidos en volúmenes seguirían a salvo en dicho volúmen.

2.2.4.c. Elementos alrededor de las *VMs*

Como se ha adelantado ya, existen multitud de elementos necesarios alrededor de las *VMs* para poder componer una solución.

Un ejemplo es el caso de las redes. Normalmente son redes definidas por software. Se definen lógicamente como se comunican las diferentes *VMs* entre ellas: rangos de IPs, puertos permitidos, *DNS* interno e incluso parámetros de tráfico como *QoS*[27]. También se define la interfaz de este "mundo virtual" con el exterior: *DNS* externo, rango de IPs, puertos abiertos.

Otro elemento importante que ya se ha mencionado son los volúmenes de datos. La filosofía del *cloud* es que todo es efímero excepto lo que se persiste en un volúmen. Es común hacer que las *VMs* tengan el sistema operativo en un disco efímero, y sobre el sistema de archivos se monten volúmenes persistentes donde se vayan a guardar datos que se quieran persistir. En ocasiones es complicado separar el estado de una aplicación (datos que se quieren persistir), por lo que es una práctica llevada a cabo el montar todo el sistema de archivos en un volúmen persistente.

2.2.4.d. Proveedores de *plataformas de VMs*

Se pueden separar estos proveedores en dos mundos: el privado (soluciones que se pueden montar en datacenters privados o *in-house*) y el público (soluciones que se venden como servicio en las que el proveedor es el responsable de los datacenters).

En el contexto **privado** hay dos grandes combatientes.

En el mundo *open-source* tenemos a ***OpenStack***[28]. Cualquiera con suficientes conocimientos podría comprar el hardware adecuado, instalar *OpenStack* y ofrecer *VMs* a través de una API. *OpenStack* es ampliamente usado en administraciones públicas de toda Europa, como por ejemplo en la *Junta de Andalucía*.

[27] *Quality Of Service*
[28] www.openstack.org/software.

Fuera del mundo *open-source*, es decir, como solución privativa, quizá el ejemplo más común sea la tecnología de ***VMware***. *vSphere*[29] es una suite de soluciones para instalar y gestionar una plataforma de *cloud computing* sobre un datacenter. Muchas de las empresas que ofrecen servicios de *VPS*[30] usan esta tecnología por debajo. También es bastante común su uso en servicios internos en grandes compañías.

En el contexto de cloud **público** hay muchos competidores.

Por un lado está el pionero, ***Amazon Web Services***[31]. Amazon parece dominar el mundo del cloud público. Tiene una oferta consolidada tras más de 10 años. Cuenta con clientes enormes, como *Netflix*. Muchas grandes compañías confían sus *workloads* a *AWS*.

Como parte de un segundo plano, se tiene a ***Google Cloud***[32]. Google usa una tecnología interna muy avanzada para prestar sus servicios, *Borg*[33]. Parte esa tecnología se usa para su oferta de cloud.

En este segundo plano hay también otro gigante, *Microsoft*. *Azure*[34] es la apuesta de *Microsoft* para ofrecer servicios de cloud. Esta parece ser la evolución natural de muchas grandes compañías que quieren prescindir de sus propios datacenters y que ya están usando otros productos de *Microsoft*.

Además de estos tres grandes proveedores de cloud públicos, hay muchos otros proveedores, pero su tamaño e importancia es significativamente inferior. Cabe mencionar que muchos de estos pequeños proveedores usan *OpenStack* para ofrecer sus servicios.

Existe también el término *hybrid cloud*. Suele usarse para referirse a soluciones mixtas de cloud que tiene parte privada (en data centers *in-house*) y parte pública (usando servicios como *AWS*). Esta combinación es muy común como transición hacia la cloud pública en empresas grandes.

2.2.5. Nueva generación de *cloud computing*: *containers*

En el mundo de la computación y los sistemas operativos hay una nueva tendencia, los *containers*.

[29] www.vmware.com/products/vsphere.html.
[30] *Virtual Private Server*
[31] aws.amazon.com.
[32] cloud.google.com.
[33] research.google.com/pubs/pub43438.html.
[34] azure.microsoft.com.

2.2.5.a. Definición de *container*

A día de hoy no hay un criterio formal para definir qué es un *container*. Hay varias tecnologías en esta familia, con sutiles y no tan sutiles diferencias entre ellas.

Ninguna de las tecnologías que aquí se tratarán son tecnologías de virtualización *per se*, en el sentido en el que ninguna de ellas virtualiza hardware para hacer correr un sistema operativo sobre el. Es más apropiado ver a un *container* como una "partición" dentro del sistema operativo.

De hecho, un *container* puede ejecutarse tanto sobre hardware virtualizado como sobre *bare metal*[35].

Es un mismo *kernel* el que comparten todos los containers. Ese *kernel* dispone de varias características que permiten dividir los recursos entre procesos, de forma que entre estos existan ciertas barreras o límites virtuales.

2.2.5.b. Características del *kernel* que habilitan a los containers

Este apartado cubre las características del *kernel* de *Linux*. Otros sistemas operativos tienen características similares, pero no se cubrirán en este documento.

2.2.5.b.a. *Cgroups*

Es la abreviación de *control groups*. Permite que el kernel establezca límites de recursos (CPU, memoria, I/O en disco, uso de red, etc) sobre un conjunto de procesos.

Un *cgroup* es una colección de procesos que están regidos por el mismo criterio en cuanto a límites en recursos.

Estos grupos de procesos son, además, jerárquicos. Cada grupo hereda los límites del padre (a no ser que explícitamente se modifiquen).

[35] *bare metal* es un término usado para referirse a máquinas (computadoras) no virtuales, sino máquinas físicas reales.

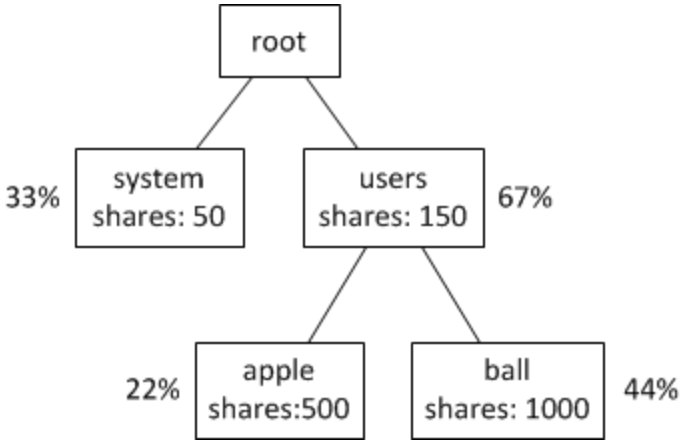


Fig 3. Concepto de división de recursos en procesos.

El *kernel* proporciona acceso a múltiples controladores (*subsystems*) a traves de la API de los *cgroups*. El siguiente gráfico muestra las particiones de recursos en función de diferentes *cgroups*.

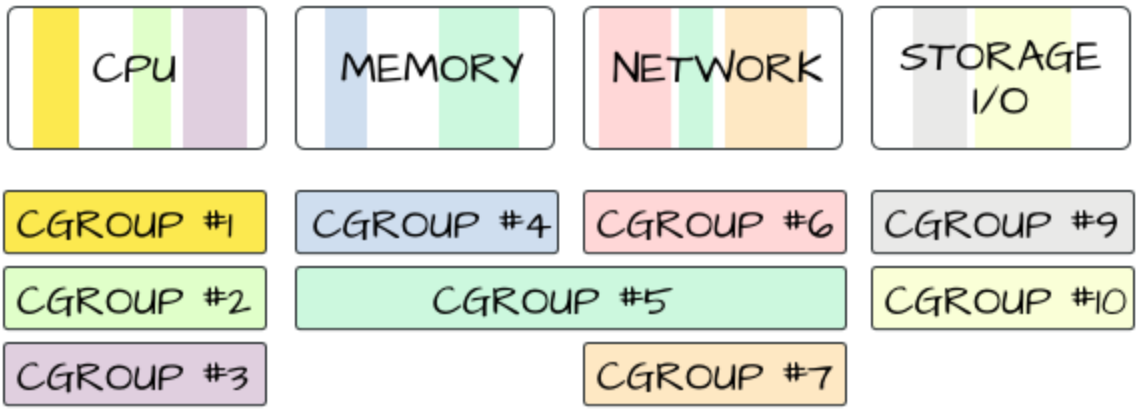


Fig 4. División de distintos tipos de recursos según *cgroups*.

2.2.5.b.b. *Namespaces*

Esta característica del kernel *Linux* permite aislar recursos del sistema para una colección de procesos.

En este casos hablamos de recursos como *PIDs*[36], *hostnames*[37], espacio de *UID*[38]/*GID*[39], interfaces de red, *IPCs*[40] y, por supuesto, sistemas de archivos.

Si al lector le es familiar el concepto de *chroot*, esto se le parece bastante. Al igual que en un *chroot*, el proceso tendría una visión del sistema de ficheros limitada dentro de un *namespace*. Al igual que con el sistema de ficheros, lo mismo se puede lograr para todos los recursos mencionados anteriormente.

Los *cgroups* pueden ser también propios de un *namespace*. Es decir, un *namespace* podría definir sus propios *cgroups*.

2.2.5.b.c. *Container runtimes*

Se ha mencionado que los *containers* no son virtualización de hardware. Sin embargo, a veces se usa el término *operating-system-level virtualization* para referirse a ellas.

Existen otras características de *Linux* que son usadas para hacer *containers* (véase *Linux Capabilities* [41] o *Linux Security Modules*[42]). Cada sistema operativo tiene algunas características propias.

Un *container runtime* es un conjunto de librerías y utilidades que usa estas características del sistema operativo para ofrecer una *API* con la que es posible crear algún tipo de *container*. Según el sistema operativo, a los *containers* también se les suele llamar *jails* (*FreeBSD*) o *zones* (*Solaris*).

Se podría decir que ***chroot*** fue el primer *container runtime*. Apareció en 1982, existe en todos los sistemas operativos *UNIX-like*, pero lo único que permite es un aislamiento parcial del sistema de ficheros. Pese a su antigüedad, sigue siendo popular y usado hoy día.

Entre 2000 y 2008 aparecieron varios proyecto más avanzados para crear entornos que podríamos llamar *containers*. ***Virtuozzo*** (año 2000) fue uno de estos proyectos y más tarde fue uno de los contribuidores de ***LXC***. ***Linux-VServer*** y ***OpenVZ*** también fueron proyectos que facilitaban la creación *containers* en Linux.

---

[36] *Process Identifier*, número de identificación de un proceso en un sistema *UNIX-like*.
[37] Nombre de un nodo en una red.
[38] *User Identifier,* número de identificación de un usuario dentro del sistema.
[39] *Group Identifier*, número de identificación de un grupo de usuarios dentro del sistema.
[40] *Inter-Process Communication*, sockets, colas de mensajes, zonas de memoria compartida o señales.
[41] http://man7.org/linux/man-pages/man7/capabilities.7.html
[42] https://www.kernel.org/doc/html/v4.14/admin-guide/LSM/index.html

Podría entenderse ***LXC*** (proyecto cuya primera versión data de 2008) como la semilla del boom de esta tecnología. Estuvo impulsado por grandes compañías como *Google* e *IBM*. *Google* usa internamente containers para correr todos sus servicios. Parte del expertise conseguido se ha materializado en aportaciones a la comunidad *Linux*.

LXC ofrece una *API* relativamente fácil de usar que permite crear un entorno similar a una instalación Linux normal, pero sin la necesidad de un kernel propio.

El proyecto ***LXC*** se compone de la librería del sistema *liblxc*, diferentes utilidades *CLI*[43] para el control de los containers y *bindings*[44] para varios lenguajes de programación.

A diferencia de ***OpenVZ*** y ***Linux-VServer***, ***LXC*** si ha tenido una notable aceptación presumiblemente gracias a que *Ubuntu*, una de las distribuciones Linux más extendida, da soporte nativo para ***LXC***.

Más recientemente, en 2013, apareció ***docker***. Sin duda, ***docker*** ha sido el fenómeno que ha hecho que los *containers* sean una de las tecnologías que están revolucionando el mundo de la computación, y que cambiarán la forma en la que los desarrolladores desarrollan software y la forma en la que el software se opera en producción.

El proyecto ***docker*** ha evolucionado a un ritmo frenético. *Docker Inc.*, la compañía detrás de ***docker*** ha decidido dividir el proyecto en *docker-ce* (*Community Edition*) y *docker-ee* (*Enterprise Edition*). Siempre que se haga referencia en este texto a *docker* se está referenciando a *docker-ce*, la parte totalmente *open-source* (licencia *Apache 2*).

A la sombra de Docker surgieron otros proyectos como ***lmctfy***[45], de la propia *Google*, o **rocket**, de *CoreOS*.

El caso de **rocket** (o **rkt**) como pasó a llamarse después, es digno de mención. Es bastante similar a **docker**, pero mientras *docker* necesita un demonio corriendo todo el tiempo para gestionar los containers, **rkt** es una *one-time-operation*. Es un simple binario que invoca a otro proceso a la vez que configura y organiza parámetros del kernel para aislarlo de distintas formas (mecanismos ya vistos en secciones anteriores).

[43] *Command Line Interface*
[44] En este contexto, *binding* hace referencia a una forma de hacer posible el uso de funciones de un lenguaje de programación o primitivas de un *runtime* dentro de otro lenguaje de programación.
[45] *Let Me Contain That For You*

2.2.5.b.d. *Docker*

El runtime de containers llamado ***docker*** (*docker-ce*) está compuesto de varias tecnologías. Realmente cuando se habla del proyecto *open-source* se refieren a *moby*[46]. *Moby* se compone de muchos otros componentes que tienen entidad propia y que incluso con intercambiables.

Docker descansa sobre las bases de ***LXC***. Ocurre que no todos los sistemas tienen las características necesarias (*cgroups*, *namespaces,* etc) para soportar ***LXC*** y, por tanto ***docker***. En *Docker (Inc)*, crearon *LinuxKIT*[47] con el objetivo de ofrecer un subsistema Linux mínimo sobre cualquier plataforma, para habilitar su tecnología de containers sobre (prácticamente) cualquier arquitectura.

LinuxKIT puede ejecutarse sobre *Windows*, *macOS*, o incluso sobre *bare metal*[48]. Esto hace pensar que podría ofrecerse un servicio similar a un cloud, pero con containers.

Sobre *LinuxKIT*, o sobre prácticamente cualquier distribución *Linux* moderna, se ejecutaría el propio runtime, *containerd*[49]. *Containerd* es ahora un proyecto separado que pretende ser el runtime estándar para containers en la industria. A día de hoy, todo parece indicar que tiene el rumbo correcto, pues es un proyecto aceptado en la *CNCF*[50]. De hecho *Docker Inc.* lo donó con el propósito de sentar las bases de este estándar, lo que también dió origen a la *OCI*[51], que es ahora un proyecto bajo el paraguas de *The Linux Foundation*[52].

De ahora en adelante, cuando se hable de *containers* en este documento, inherentemente se está refiriendo a *docker containers*.

---

[46] https://mobyproject.org/
[47] https://github.com/linuxkit/linuxkit
[48] En este contexto *bare metal* se refiere a ejecutar algo sobre un hardware directamente, sin sistema operativo de por medio.
[49] https://containerd.io/

[50] Cloud Native Computing Foundation, https://www.cncf.io/
[51] Open Container Initiative, https://www.opencontainers.org/
[52] https://www.linuxfoundation.org/projects/

2.2.5.c. Problemas que solucionan los *containers*

2.2.5.c.a. Acoplamiento software-hardware

Anteriormente se ha explicado con detalle el problema del acoplamiento entre software y hardware. Se ha visto también que las máquinas virtuales mitigan, en cierto modo, ese problema pero, en ocasiones, su eficiencia es, cuanto menos, dudosa. Particionar los recursos mediante máquinas virtuales puede ser verdaderamente un derroche de recursos "malgastados" en virtualización. Si se quiere virtualizar un proceso, puede que el coste en huella de memoria, disco y proceso del sistema operativo virtualizado sea muy superior al del proceso o conjunto de procesos que se quiere virtualizar. En esto los containers son mucho más eficientes que las máquinas virtuales. En un container, idealmente, podríamos meter únicamente un el binario del proceso o procesos que queramos ejecutar y las librerías necesarias (dependencias). Habría que super, eso sí, el coste del runtime de containers, pero este es el compartido entre todos los containers.

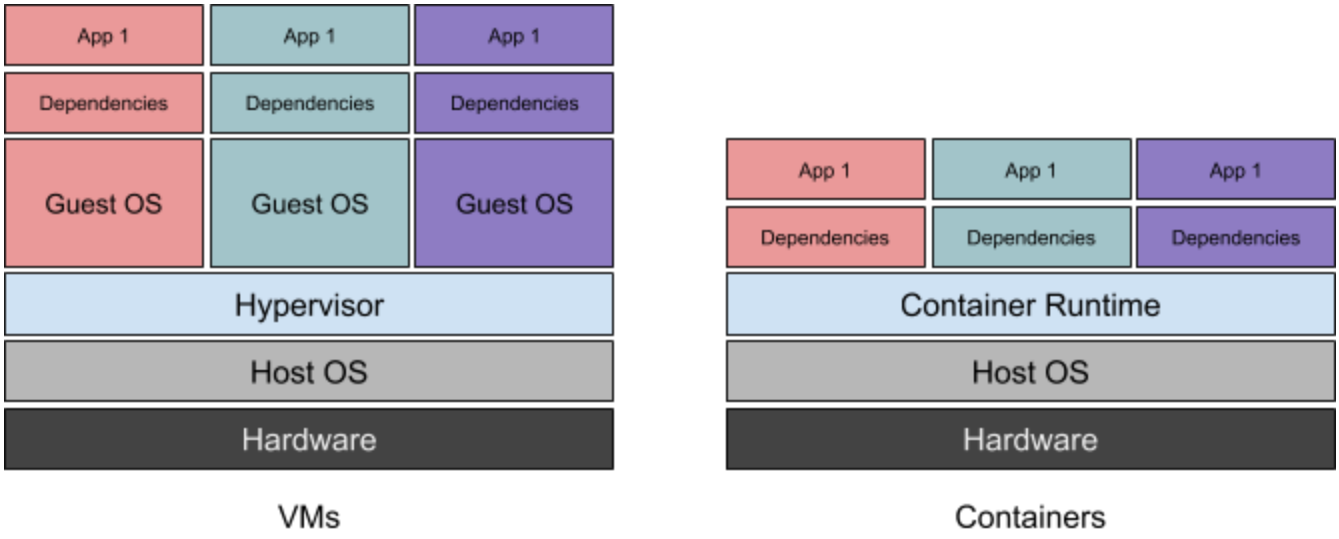


Fig 5. Comparativa: *VMs* vs *containers*.

Cabe mencionar que, en ocasiones, los containers parten de una imagen base que es parte de un sistema operativo. Esa imagen contiene simplemente algunas librerías, utilidades comunes y un *runtime* mínimo.

2.2.5.c.b. Distribución de software

Otro problema que se ha tratado es la distribución y operación de aplicaciones. Cada aplicación tiene una forma óptima de ser instalada, configurada y operada, y muchas de esas formas son totalmente diferentes entre sí. Este problema es equivalente al problema de transportar mercancías cuando se empezó a desarrollar la navegación. Transportar cerveza requería unos métodos y equipamiento totalmente distintos de los necesarios para transportar grano. La solución a ambos problemas llegó con los *containers*. Los *containers* ofrecen una forma común de operar. La complejidad particular de cada carga, es responsabilidad del *container* en sí, pero del container hacia fuera, todos son iguales.

Con *containers*, desplegar una aplicación *Java* es totalmente igual a desplegar una aplicación *PHP* o desplegar una base de datos *MySQL*. Una vez se entienden las abstracciones, toda la complejidad está contenida en la imagen del container, que debe ofrecer una forma fácil, o al menos factible, de configurar su carga útil.

2.2.5.d. Morfología de un *container*

Un *container* es una instancia de una imagen del sistema de archivos en la cual se ejecutan uno o más procesos, que tienen una visión limitada de los recursos de la máquina sobre la que se están ejecutando.

2.2.5.d.a. Imagen

En *docker*, una imagen no es más que varias capas de ficheros superpuestas una encima de otra. Cada capa tiene un *hash* asociado, lo cual facilita algunas operaciones al ser posible identificar si existen cambios en las capas entre dos imágenes.

Supóngase que en unas primera capa se añade un archivo `/opt/bar` de 3 MB. Ahora supóngase la creación de una segunda capa en la que se elimina `/opt/bar` y se crea `/opt/foo`, de 2 MB.

En el container resultado de instanciar esa imagen el sistema de archivos tendría únicamente `/opt/foo`, puesto que las capas se "proyectan" una sobre otra. Sin embargo, en la imagen original existen ambas capas por separado. Es tanto así que el peso de la imagen sería de 5 MB, la suma de ambas capas.

Las imágenes también tienen capas de metadatos. Estas capas contienen información como el comando que se debe ejecutar al arrancar el container, o etiquetas arbitrarias que se pueden añadir a la imagen y que permiten identificar y gestionar esas imágenes en diferentes tipos de inventarios.

2.2.5.d.b. volúmenes

Instanciar una imagen es crear un container a partir de ella. Crear un container de *docker* también implica ejecutar un proceso que tiene como entorno ese sistema de archivos.

Dicho sistema de archivos no viene íntegramente de la imagen. *Docker* permite montar directorios especiales que están persistidos en un volúmen.

Los directorios que están fuera del volúmen son efímeros. Su ciclo de vida está íntimamente ligado al ciclo de vida del container. Cuando el container se elimine, estos archivos serán también eliminados.

Los volúmenes sin embargo son persistidos. Los volúmenes son una de las formas de interacción del container con el "mundo exterior". Por ejemplo, es posible mapear directorios internos del container con directorios del sistema host. La abstracción de ese mapeo es un volúmen.

Una aplicación debe guardar su estado en volúmenes. De lo contrario su estado se perdería o corrompería al desaparecer el container.

El concepto de volúmenes no es nuevo en los *containers*. Es una práctica común, o al menos recomendable, usar volúmenes en clouds basados en máquinas virtuales.

2.2.5.d.c. Red y puertos

Otro de los elementos centrales en el funcionamiento de las aplicaciones de servidor son las comunicaciones de red. Es normal que los containers se encuentren aislados a nivel de red.

En *docker*, por ejemplo, es posible crear redes virtuales que permiten la comunicación entre diferentes containers y también hacia el mundo exterior. Estas redes están implementadas con *Linux Bridges*[53].

[53] Este artículo explica de forma simplificada cómo funcionan los *Linux Bridges*
https://goyalankit.com/blog/linux-bridge

2.2.5.d.d. Contenido de un *container*

Véase una imagen de una aplicación típica, por ejemplo, *MySQL*. Esta imagen seguramente contendría una parte reducida de las utilidades de una distribución, los binarios propios de *MySQL*, librerías dependencias de *MySQL* necesarias para su funcionamiento, sus archivos de configuración y algunos elementos adicionales comúnmente denominados *glue code*.

Este *glue code* se encarga de facilitar la operación y configuración de la aplicación a partir de los mecanismos de los que dispone el sistema que orquesta los contenedores.

Típicamente, y para aplicaciones sencillas, la configuración se hace mediante variables de entorno: eg.: usuario y contraseña con los que configurar *MySQL*, nombre de la base de datos principal a crear y puerto en el que el servidor de la base de datos va a escuchar.

El *glue code* se encarga de leer esos parámetros y hacer que esa configuración ocurra.

2.2.5.e. Orquestación de *containers*

Para operar servicios que corren sobre container es necesaria una capa adicional sobre el runtime de containers.

Esta capa se encargaría de gestionar cómo se ejecutan los *containers* sobre las diferentes máquinas del clúster. Es responsabilidad del orquestador el configurar los nodos para que los containers tengan los recursos adecuados, puedan comunicarse correctamente y los servicios se mantengan corriendo.

El sistema de orquestación debe de tener un plano de gestión, utilizado por las personas encargadas de la infraestructura, y también un plano con el que interactúan los operadores de las aplicaciones y servicios.

Actualmente son tres los principales orquestadores para contenedores.

Docker Swarm es la apuesta de *Docker Inc.* para correr cargas sobre containers en producción. Proporciona características como soporte de escalado horizontal y balanceo de carga entre réplicas, red multi-nodo, descubrimiento de servicios mediante metadata y una sencilla API declarativa. *Swarm* nunca ha llegado a tener mucha tracción, a pesar de sus últimas mejoras. Tanto es así que incluso *Docker Inc.* ha decidido incluir soporte nativo para *Kubernetes* en sus distribuciones de

docker. Esto quiere decir que cuando los desarrolladores instalan *docker* en sus equipos para desarrollo, tienen la posibilidad de ejecutar también *Kubernetes*[54].

Kubernetes es un proyecto nacido en la cuna de *Google*. En *Google*, todo corre sobre containers. *Borg* es la plataforma que orquesta todos esos containers. *Kubernetes* es un proyecto *open-source* inspirado en *Borg* que se está convirtiendo en el estándar *de facto* en la industria. *Kubernetes* cuenta con todas las características mencionadas para *swarm*. También incluye bastantes más tipos de recursos (*LoadBalancers*, *Secrets, ConfigMaps,* y otras muchas abstracciones que permiten crear, modificar y eliminar recursos en el cloud de forma declarativa). Además cuenta con una *API* mucho más flexible. Tanto es así que se puede extender declarando e implementando nuevos tipos de recursos personalizados. Se verá en la sección de *serverless* un ejemplo de esto último. Cabe mencionar también que, a diferencia de *swarm*, *kubernetes* puede funcionar con diferentes runtimes de containers, aunque *docker* es el más común.

Por último ***OpenShift***. Es la apuesta de *RedHat* para estar presente en el mundo de los *containers*. Podría decirse que *OpenShift* es una capa encima de *Kubernetes*. Prácticamente todas las funcionalidades de *Kubernetes* están en *OpenShift* y se usan de la misma forma. Además hay algunas otras facilidades incluidas por *RedHat* (como ejemplo, antes de que *Kubernetes* incluyera un *dashboard* de visualización y control, *OpenShift* ya tenía uno). Los containers que se ejecutan en *OpenShift* tienen algunas restricciones adicionales, como que no pueden funcionar con el usuario *root*. La ventaja de *OpenShift* es que al estar soportado por *RedHat*, una compañía estandarizada en ese ecosistema lo tendrá bastante fácil para dar adoptar *OpenShift*.

2.2.6. Futura generación de *cloud computing*: *serverless*

2.2.6.a. Definición

Existe otra reciente tecnología que está empezando a ganar tracción, *serverless*. Un servicio *cloud serverless* ofrece la posibilidad de crear funciones individuales (rutinas) en algún lenguaje de programación. Estas funciones son definidas con un contexto: *runtime* a usar, librerías, etc. Por otro lado se pueden definir eventos: una petición *HTTP*, un mensaje en una cola de mensajes, un email… Se puede definir un *trigger* que haga que la función se ejecute cuando el evento ocurre. Estas funciones podrían también conectarse a otros recursos, como un almacenamiento en el *cloud* (algo

[54] https://blog.docker.com/2017/10/docker-for-mac-and-windows-with-kubernetes-beta/

del tipo *AWS S3*) o una base de datos. Las funciones pueden realizar todo tipo de acciones (al fin y al cabo son código normal), incluso reaccionar al evento (i.e. respuesta a la petición *HTTP*).

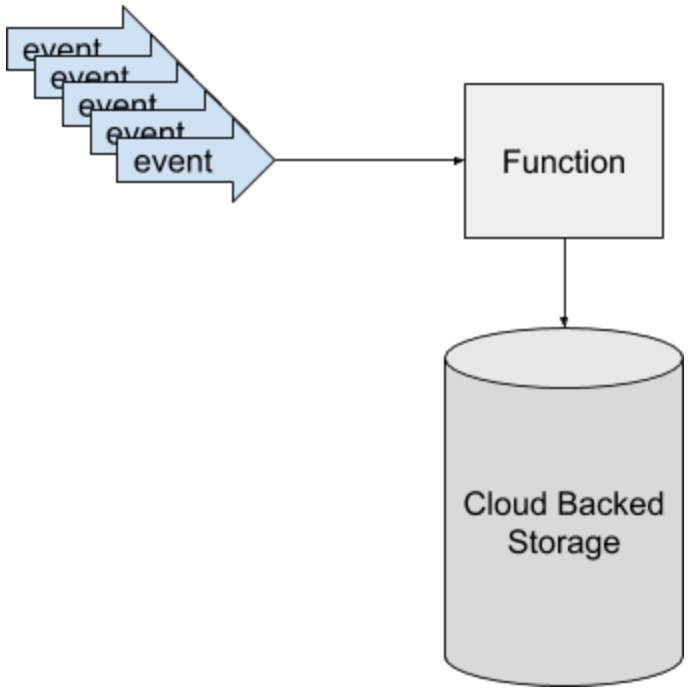


Fig 6. Caso de uso *serverless* con eventos y almacenamiento.

La primera ventaja evidente de esta arquitectura es que, desde el punto de vista del desarrollador de la aplicación, la infraestructura es inexistente y, por tanto, se evitan todas las responsabilidades relacionadas. En otras palabras, **hardware y software están totalmente desacoplados**. Por otro lado esto hace que el servicio sea, virtualmente, escalable infinitamente. Siempre que la aplicación (ahora distribuida en funciones) maneje la concurrencia en el acceso a su estado (si existiera) de forma correcta, el servicio escalaría sin hacer nada. Sería responsabilidad del proveedor de cloud tener recursos suficientes para ejecutar las funciones cada vez que sea necesario.

Este paradigma presenta algunos problemas. Para empezar, se hace complicado implementar soluciones basadas en *serverless* para ciertos problemas, sobre todo para aquellos que requieren de un amplio contexto compartido en toda la aplicación.

Por otro lado, los proveedores públicos de cloud tarifican estas soluciones en cuanto al número de ejecuciones de la función y tiempo que tarda en hacerlo. Esto implica la necesidad de tener un gran control de costes y tenerlos en cuenta en el diseño. Un simple cambio aparentemente inofensivo

puede resultar en funciones ejecutándose millones de veces de forma innecesaria causando un gran coste.

Por su naturaleza, este tipo de sistemas están siendo actualmente usados como punto de integración entre servicios. Es común que en grandes compañías haya varios grandes servicios que se han ido añadiendo con el tiempo. Es natural que ellos estén basados en diferentes tecnologías. Es relativamente fácil integrar servicios de naturaleza muy diferente con *serverless*. Por ejemplo, sea el caso de un servicio de almacenamiento de datos y un de mensajería interna. Los servicios de almacenamiento de datos suelen tener la posibilidad de lanzar eventos cuando archivos son añadidos, eliminados o modificados. Estos eventos pueden ser recogidos por una función que usara la *API* del servicio de mensajería para enviar un mensaje a los usuarios miembros de la carpeta compartida en cuestión.

2.2.6.b. Soluciones *serverless*

Algunos clouds públicos tienen disponibles soluciones *serverless*. El pionero fue *AWS*, con su servicio *Lambda*[55], anunciado en 2014[56]. Quizá este servicio sea incluso anterior al término *serverless*. *Google* lanzó en 2016 *Google Cloud Functions*[57], que tuvo su respuesta por parte de *Microsoft Azure* con *Azure Functions*[58] el mismo año.

También es posible tener soporte para *serverless* en infraestructura propia, especialmente si ya se tiene un clúster de *Kubernetes*. Véase el proyecto *kubeless*[59], creado por *Bitnami*.

A la par que estas soluciones se empezaron a popularizar, se crearon diferentes herramientas que ayudan a gestionar servicios basados en *serverless* de forma eficiente. El mejor ejemplo de ello es el proyecto *serverless*[60]. Este proyecto es un *framework* que permite, de una forma sencilla y uniforme, desplegar funciones en muchos de los proveedores de cloud.

[55] https://aws.amazon.com/lambda/
[56] https://sdtimes.com/amazon/amazon-introduces-lambda-containers/
[57] https://cloud.google.com/functions/docs/
[58] https://azure.microsoft.com/en-us/services/functions/
[59] http://kubeless.io/
[60] https://serverless.com/

3. Especificaciones

Se define como objetivo de este proyecto el desarrollar una solución "*cloud native*" agregador de datos *IoT*.

3.1. *Cloud Native*

El término "*cloud native*" es ya común en la industria, e implica que la solución está diseñada desde el origen para aprovechar las características del cloud computing y para lidiar correctamente con sus limitaciones. En concreto este proyecto se va a desarrollar sobre tecnología de containers. El hecho de adoptar esta tecnología condiciona totalmente el diseño de la solución, pero a cambio permite gestionar de una forma mucho más efectiva su operación, así como asumir grandes cargas desde el exterior.

3.1.1. El reto de funcionamiento bajo un ciclo de vida agresivo

Las soluciones de orquestación de containers imponen un ciclo de vida bastante agresivo a dichos *containers*. El orquestador puede terminar un *container* en cualquier momento y arrancar otra instancia de la misma imagen en el mismo nodo u en otro diferente. Esto puede ocurrir por multitud de causas: reparto de cargas en los nodos, un nodo va a quedar fuera de servicio para ser actualizado, un nodo está sobrecargado y otro tiene demasiados recursos libres o simplemente porque la aplicación está siendo escalada horizontalmente (durante dicho proceso el orquestador crea o elimina réplicas de los containers que sirven esa parte de la aplicación).

La aplicación debe estar preparada para funcionar adecuadamente con este ciclo de vida. Para esto hay dos cosas fundamentales: la aplicación **debe cargar rápido** y también **debe gestionar bien ser parada abruptamente**.

La aplicación debe tardar muy poco desde que se empieza a ejecutar hasta que está plenamente operativa. Normalmente las aplicaciones requieren cierto tiempo de inicio para leer configuración, hacer comprobaciones, conectar con diferentes recursos de los que dependen, o simplemente cargar librerías y dependencias en memoria. Muchas de esas operaciones son imposibles de optimizar, pero otras pueden hacerse de forma mucho más eficiente en estos entornos. Es el caso de las comprobaciones o *health checks*. Normalmente es posible descargar a la aplicación de la

responsabilidad de hacerlas al empezar, y se pueden definir comprobaciones que son ejecutadas por el orquestador. Por ejemplo, algunas aplicaciones tienen mecanismos internos para comprobar que tienen conectividad con la base de datos. En lugar esperar a que esos mecanismos actúen, se puede definir un *trigger* que ejecuta dicha prueba a petición del orquestador. De esta forma el orquestador puede activamente hacer esa comprobación y tomar decisiones en función del resultado sin esperar a que ocurra por el ciclo de vida natural de la aplicación.

Como se ha adelantado, la aplicación debe también gestionar bien paradas abruptas. Es común que el orquestador "mate" los *containers* frecuentemente por diferentes motivos. Para esto el orquestador mandará una señal *SIGTERM* (esto realmente depende del orquestador, pero es el comportamiento más común). Al recibir esa señal, el proceso debería liberar todos los recursos y terminar las operaciones atómicas que hubiera comenzado. También es común que si en un tiempo razonable el proceso no ha terminado, el orquestador mande la señal *SIGKILL* que terminaría de inmediato con el proceso.

3.1.2. El reto del paralelismo

Una las características más atractivas de los orquestadores de *containers* es lo fácil que puede ser escalar horizontalmente una aplicación. Para que eso ocurra, la aplicación debe estar preparada y gestionar correctamente que varias instancias del proceso se estén ejecutando en simultáneamente.

Para esto es fundamental que el acceso a los datos esté bien sincronizado. A veces esto simplemente se delega a una base de datos. Otras veces eso no es posible y se requiere el uso de semáforos de acceso y otros mecanismos para permitir que las diferentes instancias usen los datos ordenadamente y nunca se pierda la consistencia.

Cuando hay varias réplicas de un mismo proceso (cada una es un container), suele haber un elemento extra llamado balanceador de carga. Este elemento se encarga de repartir las peticiones entrantes entre las diferentes réplicas.

Según el tipo de aplicación, puede ser importante que las peticiones correspondientes a una misma sesión siempre vayan a la misma réplica. Algunos balanceadores permiten configurar afinidad de sesión.

3.1.3. Mantener estado consistente

El estado de una aplicación es todo el conjunto de datos que no son la aplicación en sí. Esto es datos de usuario, archivos temporales o por ejemplo datos generados.

Es de vital importancia que si esos datos son importantes, no se guarden en el sistema de archivos del *container*, pues este es efímero. Cabe recordar que el *container* puede ser reemplazado por otro nuevo en cualquier momento. Para que una aplicación funcione correctamente en este entorno debe guardar todo el estado en volúmenes de datos o alguna base de datos externa.

3.1.4. Desarrollo y despliegue

Los métodos tradicionales para desplegar software y mantenerlo ya no son válidas en estos nuevos paradigmas. No es posible hacer *SSH* a un *container* y ejecutar la nueva versión del código de la aplicación, ya que ese *container* no es más que una instancia efímera de una imagen.

El despliegue de nuevas versiones de la aplicación ha de hacerse en forma de nuevas imágenes que el orquestador tomará para reemplazar las instancias ya corriendo de la aplicación.

Por otra parte están los cambios en la configuración de la infraestructura. La forma común de usar el *cloud*, y seguramente la única forma válida si hablamos de *containers*, es mediante un sistema en el que se define el estado deseado de cada recurso y el orquestador lee ese estado, lo compara con el estado actual y hace los cambios pertinentes. Normalmente esa definición de estado deseado se puede guardar en archivos que se suelen versionar tal y como se haría con el código de la aplicación. De esa forma, es muy fácil hacer cambios en cosas como la red que sustenta toda nuestra solución, o la configuración que se da para conectar todas los componentes entre ellos.

3.2. Configuración de la infraestructura

Para poder desplegar la solución será necesario disponer de un clúster configurado adecuadamente. Esto también se considera dentro del alcance del proyecto.

Dicha tarea consiste en detallar, y automatizar en lo posible, el proceso por el cual se conectan y configuran diferentes máquinas para dar lugar a un clúster con capacidades de orquestación de contenedores.

También se incluye la instalación y configuración de todos los elementos necesarios para dar soporte a la aplicación. Esto es la capa de red virtual, roles de administración, cuentas para el *pipeline* de desarrollo, interfaz con el exterior, provisión de volúmenes de datos, y cualquier otro elemento que sea requerido para que la solución pueda ser desplegada, actualizada y usada.

3.3. *Pipeline* de desarrollo

Se incluye dentro del alcance del proyecto la creación y configuración de un *pipeline* de desarrollo adecuado que permita desplegar y actualizar la aplicación de forma automática. Esto es un subsistema que no forma parte de la aplicación directamente, sino que es algo externo que se ocupa de automatizar los diferentes procesos que han de ocurrir para que cualquier cambio que ocurre en la aplicación (en su código fuente) se aplica en la aplicación que se está ejecutando en producción.

A este subsistema se le llama comúnmente *CI/CD*[61]. Es algo que comúnmente se descuida, pero que se vuelve vital cuando se está usando este tipo de *cloud*. En otros entornos el proceso de desplegar la aplicación puede ser muy simple (tanto como copiar unos archivos a un servidor que está corriendo). Con *containers* esto se complica. Primero hay que construir las imágenes de los *containers* que van a ejecutarse. Este paso puede a su vez requerir diferentes etapas, ya que puede haber que generar artefactos intermedios con software que no se desea que esté en la imagen final. Posteriormente hay que hacer que esas imágenes estén disponibles para que el orquestador del clúster las descargue, esto es publicarlas en un *registry*. A continuación hay que cambiar la configuración del despliegue para que use las nuevas imágenes. Finalmente hay que pasar al orquestador esta nueva configuración. El orquestador se ocupará de cambiar lo necesario para que la nueva versión de la aplicación se despliegue. Es un caso simple este cambio sería simplemente sustituir los containers que están corriendo por otros que se instancian desde la nueva imagen. También se pueden cambiar otras cosas en la configuración, como parámetros de la aplicación o incluso la topología de red.

3.4. Aplicación

La aplicación como tal forma también parte del alcance. Se ha de desarrollar una aplicación que exponga un punto de acceso en el que puede recibir datos de tipo evento. La aplicación deberá procesar esos eventos y añadirlos a una base de datos. Posteriormente esos datos pueden ser

[61] *Continuous Integration* / *Continuous Delivery*

consultados y agregados de diferentes formas. La solución deberá tener también un portal donde puedan ser visualizados los eventos registrados.

Al ser esta aplicación un colector de datos IoT, deberá presentar una API CoAP. CoAP es el protocolo por defecto para comunicaciones de dispositivos IoT vía internet entre un cliente y un servidor.

La aplicación deberá respetar los puntos detallados en la anterior sección titulada *cloud native*.

Para poder probar la aplicación, será necesario desarrollar algún cliente de ejemplo que envíe los datos que serán consumidos por la aplicación.

3.5. Tecnologías *open source*

Siendo este proyecto una investigación académica, se asume que debe haber preferencia por proyectos libres y de código abierto.

Siempre que sea posible, debería optarse por tecnologías de este tipo. Cualquier uso de aplicaciones, librerías o servicios privativos debe ser cuidadosamente justificada.

3.6. Tabla resumen de especificaciones

Se incluye a continuación una tabla resumen de las especificaciones concretas para este desarrollo.

Requisitos para considerar la aplicación Cloud Native	
1.1. Tiempo carga	El proceso principal debe arrancar en un tiempo inferior a 3 segundos cuando el proceso tiene una CPU dedicada.
1.2. Reinicio abrupto	El proceso principal gestiona la señal SIGTERM para no dejar el estado inconsistente.
1.3. Paralelismo	El proceso principal puede escalar horizontalmente (más de una réplica pueden funcionar en paralelo).
1.4. Persistencia	Todo el estado de la aplicación debe ser persistido en volúmenes.
1.5. Despliegue en base a manifiestos	La forma de desplegar la aplicación sobre la infraestructura debe ser completamente automática, siendo la orden inicial del usuario la

	única interacción.
1.6. Desacople del hardware	La aplicación no debe hacer uso directo del hardware.
Requisitos de configuración de la infraestructura	
2.1. Máquinas	Configuración de las máquinas para que sean controladas por el plano de control.
2.2. Capa de red	Debe haber una capa de red que permita la comunicación entre contenedores y con el exterior.
2.3. DNS	Debe haber un servicio interno al clúster que sirva para resolución de nombres internos.
2.4. Seguridad	Se debe asegurar que las comunicaciones entre procesos en el clúster son seguras y otros usuarios del mismo clúster no pueden intervenirlas.
2.5. Plano de control	Debe configurarse un plano de control, que es el conjunto de componentes que orquestan los nodos clúster, contenedores, servicios y demás elementos reales o virtuales.
Requisitos del pipeline de desarrollo	
3.1. Integración continua	Debe proveerse un subsistema que construya de forma automática las imágenes necesarias tras cada cambio en el código fuente.
3.2. Despliegue continuo	Debe proveerse un subsistema que despliegue las nuevas versiones de imágenes, además de los cambios de configuración de forma automática.
Requisitos de funcionalidades de la aplicación	
4.1. API externa	Debe presentar un punto de acceso por el que recibir datos del exterior.
4.2. Protocolos adecuados para *IoT*	La *API* de acceso al servicio deberá funcionar sobre protocolos adecuados para *IoT*.
4.3. Ingesta de datos	Debe ser capaz de procesar datos al vuelo y almacenarlos en una base de datos.

4.4. Visualización de datos	Debe ser posible observar los datos de forma gráfica.
Tecnologías *open source*	
5.1. Preferencia por tecnologías de código abierto	Todos los componentes, librerías y servicios usados deberán ser de código abierto a menos que haya un motivo justificado.

Fig 7. Tabla resumen de especificaciones.

4. Desarrollo de la solución

4.1. Elección de tecnología de orquestación

Para comenzar habría que tomar una primera decisión sobre qué generación de *cloud* usar: máquinas virtuales, contenedores o funciones.

Una solución basada exclusivamente en funciones no sería realmente posible, ya que es necesaria una base de datos corriendo todo el tiempo. Sin embargo, si asumimos que la base de datos va a funcionar con otra tecnología, la parte de la aplicación que es puramente ingesta de datos sí que encaja, *a priori*, dentro de lo posible en una implementación *serverless*.

Fig 8. Proyectos *serverless open-source*.

Se han investigado diferentes *frameworks* de código abierto para funciones (*kubeless*[62], *fission*[63], *open-faas*[64]). Todos ellos usan contenedores por debajo para ejecutar las funciones. Tienen un problema común, y es no pueden escalar correctamente horizontalmente, al menos no de forma fácil. Otra desventaja es que se pierde control sobre la capacidad de controlar los márgenes de recursos de los que la aplicación dispone.

Existen servicios ofrecidos por los proveedores públicos de cloud que podrían servir para realizar esta parte de la aplicación. Tanto *AWS* como *Google Cloud* ofrecen funciones como servicio (*Lambda* y *Cloud Functions* respectivamente). Estos servicios tienen un *SLA*[65] que garantiza que no haya que preocuparse por la escalabilidad. Un inconveniente de estos servicios es el coste. Normalmente se tarifica en base a cuantas veces se ejecuta la función. Para una aplicación de la que se espera que esté consumiendo

Fig 9. Servicios *serverless*.

[62] https://github.com/kubeless/kubeless
[63] https://github.com/fission/fission
[64] https://github.com/fission/fission
[65] *Service Level Agreement*

peticiones continuamente, el coste incrementa enormemente. Dado que se ha fijado como requisito optar por soluciones de código abierto, estas opciones no tienen cabida para el desarrollo de este proyecto.

En el otro extremo se encuentra la tecnología de máquinas virtuales. Sería posible desarrollar esta solución con máquinas virtuales, pero toda la parte de escalado horizontal sería mucho menos eficiente. Además ya se ha comentado que es mucho más ventajoso usar contenedores si la aplicación se diseña a conciencia para que funcione bien sobre ellos.

Por tanto, se va a diseñar la solución sobre contenedores. En el apartado sobre orquestación de *containers* en el estado del arte se ha hecho un recorrido sobre las diferentes plataformas de orquestación de *containers*. Si se compara *Docker Swarm* con *Kubernetes* y *OpenShift*, los dos últimos son superiores en todo lo relativo a gestión de los nodos del clúster. También son más avanzados en cuanto a capacidades de *SDN*[66]. Soportan una gran variedad de *CNI*[67], muchos de los cuales no son propietarios de los proveedores de cloud públicos, sino que funcionan sobre *bare metal*. *Openshift* sería la opción ideal si se tratara de un entorno *RedHat Linux*. En este proyecto no se han impuesto restricciones al sistema operativo base que usar (siempre que sea *Linux*), por tanto *Kubernetes* es una solución más adecuada al ser más genérica.

Fig 10. Logo de *Kubernetes*.

4.2. Introducción de conceptos de *Kubernetes*

Los elementos constructivos básicos en *Kubernetes* son los siguientes:

[66] *Software Defined Networking*
[67] *Container Networking Interface*

4.2.1. *Namespace*[68]

Todos los recursos que se crean en *Kubernetes* pertenecen a un *namespace*. *Kubernetes* mantiene cierto aislamiento entre los *namespaces*.

Es posible por ejemplo establecer reglas de comunicación entre *namespaces*, con lo que se podría limitar la visibilidad de servicios.

4.2.2. *Pod*[69]

Un *pod* es un elemento lógico que comprende uno o más *containers*. Es la unidad mínima de replicación y planificación.

Los containers en el mismo *pod* comparten visión de la red y tienen el mismo *hostname*. También comparten el espacio de *PID*.

Son efímeros. El *scheduler* o *planificador* del clúster puede decidir reemplazar un pod por otro con nuevas propiedades, o en otro nodo, o simplemente eliminarlo sin más.

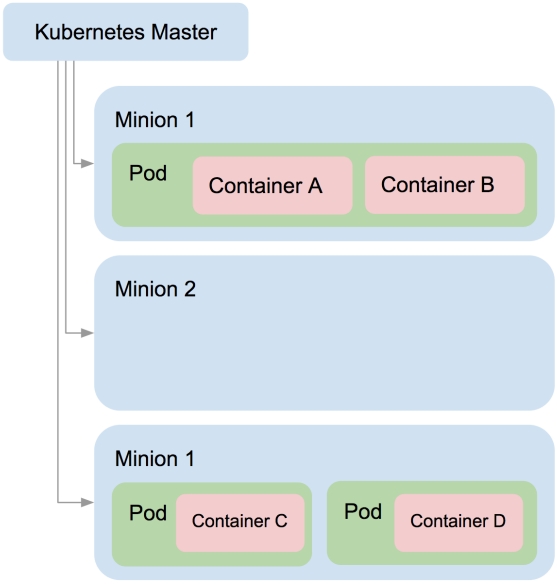


Fig 11. Distribución de *pods*.

[68] https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces
[69] https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

4.2.3. *Deployment*[70]

Un *deployment* es una entidad superior al *pod* que define ciertas propiedades external al *pod*, pero relacionadas de alguna forma.

Con la definición de un *deployment* se especifica qué cantidad de réplicas de un determinado tipo de *pod* habrá.

El *scheduler* se encargará de crear o eliminar *pods* hasta satisfacer la definición actual del *deployment*.

4.2.4. *DaemonSet*[71]

Simplificando, se podría decir que un *daemonSet* es un *deployment* con la particularidad de que se asegura que haya una réplica del *pod* corriendo en cada nodo del clúster.

4.2.5. *Volume*[72]

El sistema de ficheros de los *containers* es efímero: cuando un container desaparece, los datos desaparecen con él.

Los recursos de tipo *volume* son una abstracción que permite montar sistemas de ficheros no efímeros sobre el sistema de archivos principal del *container*. De esta forma algunos directorios puede ser persistidos y vueltos a montar en otra instancia del container.

Existen multitud de tipos de volúmenes, pero para el objetivo de este proyecto sólo se van a utilizar dos: *persistentVolumeClaim* y *configMap*.

El primero de ellos, *persistentVolumeClaim* está pensado para ser usado en conjunción con el recursos *persistentVolume*. Un *persistentVolumeClaim* define unos requisitos en cuanto a capacidad deseada y etiquetas arbitrarias adicionales. Un *persistentVolume* tiene ciertas propiedades como capacidad y otras etiquetas arbitrarias. Cuando se crea un *persistentVolumeClaim*, *Kubernetes* intenta buscar un *persistentVolume* que satisfaga los requisitos fijados. En caso de que lo encuentre, asigna ese *persistentVolume* al *persistentVolumeClaim*, lo que significa que el *pod* que monte ese

[70] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[71] https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/
[72] https://kubernetes.io/docs/concepts/storage/volumes/

volúmen puede escribir datos en un volúmen de datos en el que se garantiza la persistencia de los mismos.

El segundo tipo es más especial. Los volúmenes de tipo *configMap* no están pensados para que el container persista datos en ellos. Estos volúmenes contienen una información prefijada que se ha introducido desde un agente externo (*i.e.* un administrador del sistema). Cuando el container monta este volúmen, la información declarada en ese *configMap* es montada en forma de archivos dentro del container. Típicamente se usa para inyectar archivos de configuración.

4.2.6. *Service*[73]

Las comunicaciones directas entre pods no estás permitidas. Los servicios son la forma en que se enruta tráfico hacia los diferentes pods.

Un servicio tiene asociado uno o más puertos, y también tiene un selector que indica cuales son los pods a los que el servicio se dirige. En otras palabras: si se quiere exponer un servidor web que está corriendo en un pod escuchando el puerto 80, se deberá crear un servicio con puerto 80, y con un selector que el pod anterior satisfaga.

Existen diferentes tipos de servicios. He aquí los más interesantes para este proyecto.

Los servicios del tipo *clusterIP* son servicios que tienen asignada una IP privada dentro del rango del clúster. Es el tipo por defecto.

[73] https://kubernetes.io/docs/concepts/services-networking/service/

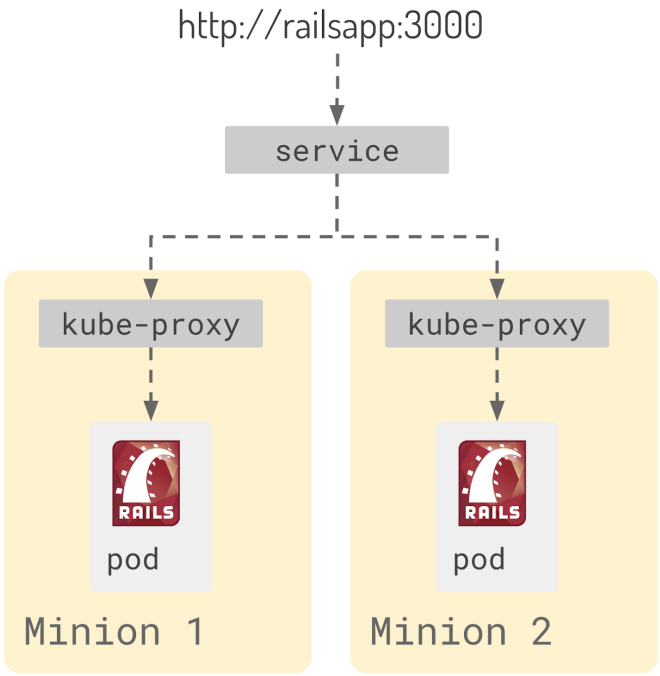


Fig 12. *ClusterIP* service.

Los servicios tipo *NodePort* son servicios que abren un puerto en concreto en todos los nodos del clúster. Este tipo de servicio suele combinarse con los ya mencionados *daemonSets*.

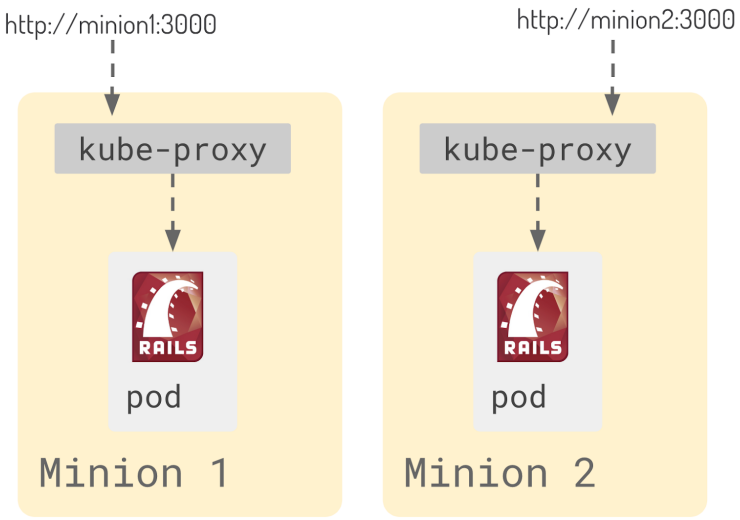


Fig 13. NodePort service.

Los servicios tipo *LoadBalancer* son servicios orientados para ser expuestos al exterior del clúster. Toman una IP (normalmente pública) de un *pool* y balancean todo el tráfico entrante entre todos los pods que satisfagan su selector.

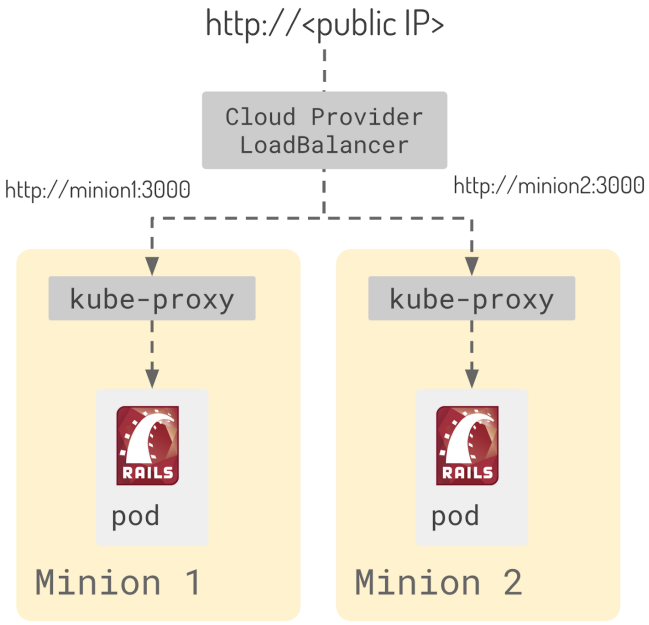


Fig 14. *LoadBalancer* service.

4.2.7 *Ingress*[74]

Los recursos tipo *ingress* son una abstracción para controlar el acceso exterior a un servicio.

Según la implementación, un *ingress* puede aportar balanceo de carga, terminación SSL y gestión de *virtual hosts*[75] y rutas de la URL.

En un *ingress* se configura por un lado cómo tratar el tráfico entrante (terminación SSL, redirecciones, reescrituras de URL, etc) y por otro lado el destino de esas peticiones, que siempre es un recurso de tipo *service*.

[74] https://kubernetes.io/docs/concepts/services-networking/ingress/
[75] En el contexto de un servidor web, se denomina *virtual host* a cada uno de los nombres de dominio que sirve el mismo servidor. Esto es, un mismo servidor web atiende por el mismo puerto a peticiones de *midominio1.com* y de *midominio2.com*. A nivel de red (nivel 6) solo hay un servidor, pero a nivel de aplicación el servidor es capaz de diferenciar el dominio destino de cada petición, ya que las cabeceras HTTP incluyen esa información. Cabe mencionar que en este ejemplo, ambos dominios resolverían a la misma dirección IP.

4.2.8. Selectores y etiquetas

Las etiquetas son parejas clave-valor arbitrarias que acompaña a cualquier objeto en *Kubernetes*. En combinación con los selectores, permiten organizar los objetos en el clúster.

Los selectores identifican a un conjunto de objetos cuyas etiquetas satisfacen su condición.

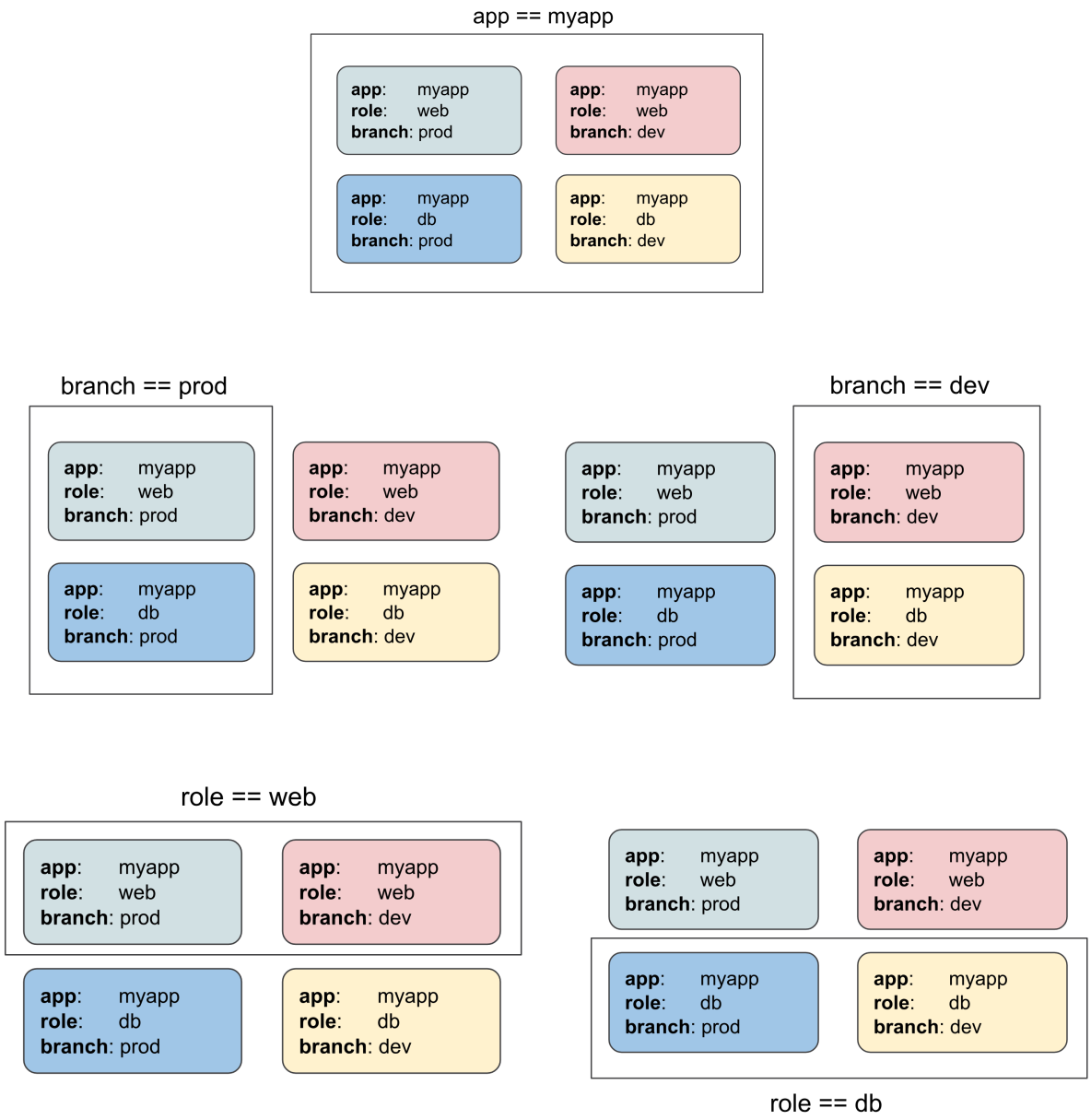


Fig 15. Ejemplo de etiquetas y selectores.

4.3. Creación de un clúster de *Kubernetes*

4.3.1. Hardware y configuración de red

Para este proyecto se cuenta con tres máquinas *HP ProLiant ML110 G5* conectadas a una red local vía Ethernet.

La siguiente tabla presenta las características principales de las máquinas y sus unidades de disco

Nombre	**moleman0**	**moleman1**	**moleman2**
CPU	Intel(R) Xeon(R) CPU 3065 @ 2.33GHz		
Memoria	8 GB	4 GB	4 GB
Almacenamiento	- `/`: 400 GB - `/media/jfc/stu0`: 1 TB	-/: 400 GB	-/: 400 GB

Fig 16. Características de nodos del clúster.

Además se dispone de un *switch* simple *TP-Link Tl-sg1008d* de 8 puertos gigabit y de un router que da acceso a internet.

Los equipos de conectan a nivel de red como describe el siguiente esquema.

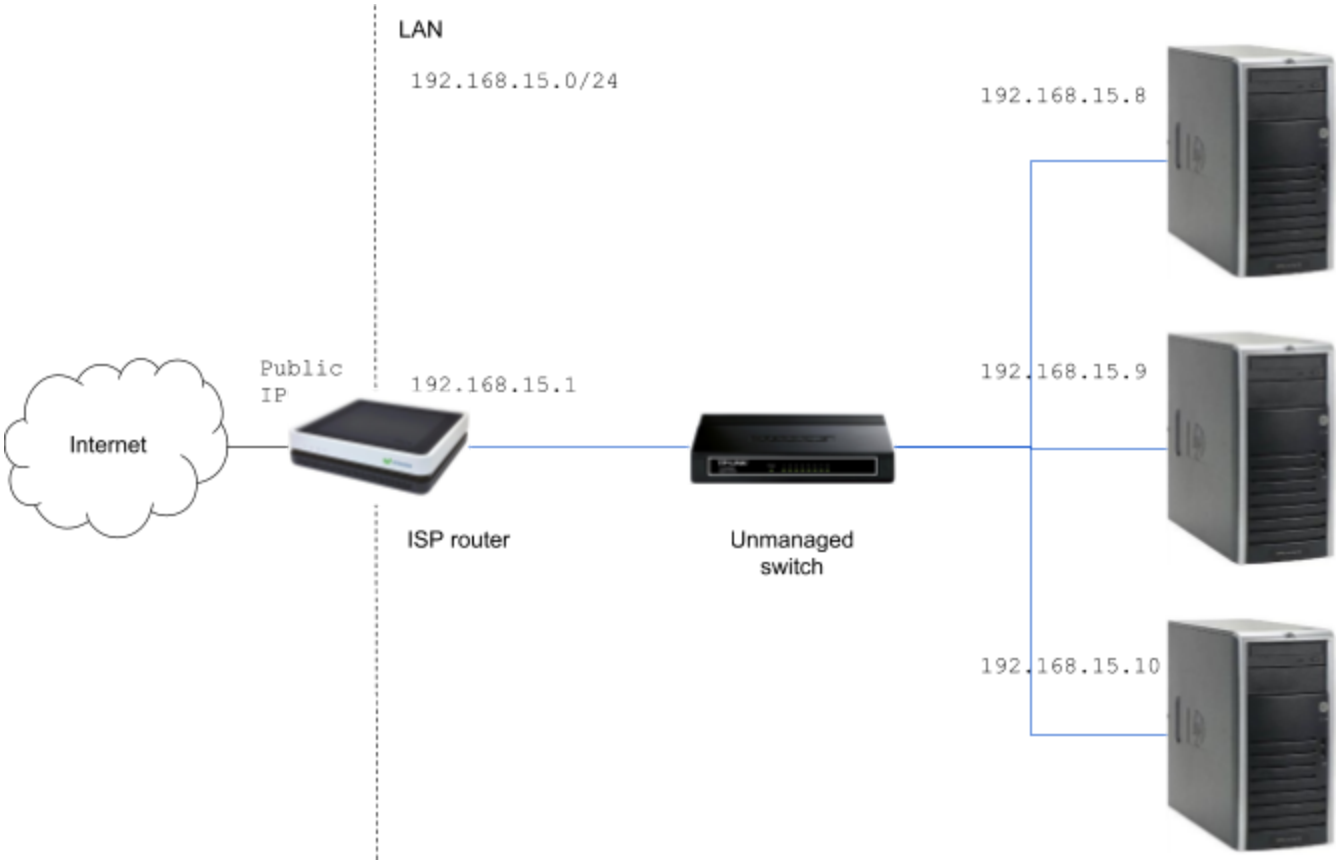


Fig 17. Diagrama de red local.

4.3.1.a. Direccionamiento

Las red local tiene el rango de direcciones `192.168.15.0/24`. La red tiene direcciones IP dinámicas asignadas por un servidor *DHCP* en el router principal. Este router es proporcionado por un proveedor de servicios de Internet, pero es posible hacer configuraciones avanzadas en el.

Por un lado se ha configurado el *DHCP* para asignar direcciones en el rango entre la `192.168.15.30` y la `192.168.15.99`. Esas direcciones son para cualquier dispositivo que se conecte a la red, excepto para los servidores, que tienen asignadas las siguientes direcciones IP según sus direcciones MAC.

moleman0	**moleman1**	**moleman2**
`192.168.15.8`	`192.168.15.9`	`192.168.15.10`

Fig 18. Direcciones IP nodos.

Nótese que se deja libre el rango entre `192.168.15.100` y `192.168.15.200`. Este rango será gestionado por *MetalLB* para proveer balanceadores de carga, como se verá más adelante.

4.3.1.b. Conexión con el exterior

Para las conexiones salientes, se hace uso de un *NAT* básico. La puerta de enlace de todos los dispositivos es el router principal (`192.168.15.1`). El router enmascara la IP privada de origen sustituyendola por la IP pública asignada en ese momento por el proveedor de internet.

Puesto que la IP pública externa es dinámica (puede cambiar en cualquier momento), se ha configurado un servicio de DNS dinámico, de forma que cualquier subdominio `*.moleman.jftechnologies.xyz` resuelve a la dirección IP pública del router.

Para las conexiones entrantes, se ha configurado *NATP* de la siguiente forma:

Puerto destino en interfaz pública	***Host* destino interno**
`80 (HTTP)`	`moleman0 (192.168.15.8)`
`443 (HTTPS)`	
`10022`[76]	
`5683 (CoAP)`	`LoadBalancer 192.168.15.100`

Server Name	External Port Start	External Port End	Protocol	Internal Port Start	Internal Port End	Server IP Address	WAN Interface
http	80	0	TCP	80	0	192.168.15.8	ppp0.1
https	443	0	TCP	443	0	192.168.15.8	ppp0.1
gitlab-ssh	10022	0	TCP/UDP	10022	0	192.168.15.8	ppp0.1
CoAP	5863	0	UDP	5863	0	192.168.15.100	ppp0.1

Add Remove

Fig 19. *Configuración NATP*.

El resto de conexiones se rechazan.

4.3.2. Sistema operativo y configuración básica de las máquinas

En las máquinas se ha instalado *Ubuntu Server 17.10*. Se ha usado una configuración básica estandar. Los equipos se conectan a la red local y toman su configuración de red automáticamente vía *DHCP*.

[76] Este puerto se usa para la conexión *SSH* contra *Gitlab* para el protocolo *GIT*. Se verá con más detalle más adelante.

El sistema de ficheros montado en *root* (directorio `/`) está soportado por un disco duro estándar de 400 GB. El equipo *moleman0* es algo especial, ya que cuenta con un disco de almacenamiento extra de 1 TB montado en el directorio `/media/jfc/stu0`. Será ese disco el que soporte los volúmenes de persistencia de todas las aplicaciones.

En todas las máquinas se ha instalado *docker-ce* y *kubeadm* siguiendo las instrucciones oficiales del proyecto. Ya se ha visto que *docker-ce* es un runtime para containers que satisface el estándar *OCI*. *Kubeadm* es una herramienta que asiste en la instalación y configuración de los componentes del plano de control de *Kubernetes*, así como en los agentes de cada nodo.

4.3.3. Instalación del plano de control de *Kubernetes*

Un clúster de *Kuberentes* no es más que un conjunto de máquinas conectadas entre sí y controladas por un sistema al que se denomina *plano de control de Kubernetes*.

El plano de control toma decisiones globales sobre el clúster (como la planificación de recursos). Es quien detecta eventos en el sistema y responde a los mismos: es quien crea una nueva réplica de un *pod* si el número de réplicas existentes no satisface la condición impuesta por el *deployment*.

Este plano de control puede ejecutarse sobre las máquinas del propio clúster o en un equipo externo [77]. Por conveniencia, para este proyecto se va a usar una de las máquinas del clúster para tal fin, *moleman0*. Esta configuración no es la más robusta para un caso real, pero es suficiente para el alcance de este proyecto.

4.3.3.a. Componentes del plano de control

Kubernetes ofrece una *API REST* que es el punto de entrada al plano de control. Esta es servida a través de *kube-apiserver*[78]. *kube-apiserver* se encarga de validar los datos de los objetos enviados a la *API*: cada llamada de tipo *REST* contiene descripciones de un recurso con multitud de parámetros, esta aplicación valida que esas descripciones tengan un esquema válido. *kube-apiserver* está diseñado para escalar de forma horizontal, por lo que pueden crearse réplicas si la carga es alta.

[77] https://kubernetes.io/docs/admin/high-availability
[78] https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver

El estado de los recursos es compartido entre todas las instancias de *kube-apiserver*. Este estado se almacena en una instancia de *etcd*[79] una base de datos distribuida de tipo clave-valor.

El ciclo de vida de todos los elementos que habitan el clúster es controlado por *kube-controller-manager* y *kube-scheduler*. El primero es un proceso con un bucle infinito que está continuamente monitorizando el estado de los recursos y asegurándose de que se coincide con el estado definido. El segundo se encarga de decidir en qué nodo ejecutar cada *pod*.

4.3.3.b. Agentes en los nodos

Cada nodo del clúster debe ejecutar varios procesos relativos a *Kubernetes*.

Por un lado está *kubelet*. Es un servicio con el que interactúa el plano de control y que sirve de interfaz para controlar los *containers* que corren que cada *pod*. En el caso de la configuración que de este proyecto, *kubelet* es la interfaz entre el demonio *docker* que corre en cada máquina y el plano de control de *Kubernetes*.

Por otro lado está *kube-proxy*, que se encarga de gestionar las reglas de *networking* (*iptables*) en el nodo y de gestionar en enrutamiento del tráfico.

4.3.3.c. Creación del nodo *master*

En primer lugar se ha de crear el nodo master, que en este caso será *moleman0*.

Para ello basta con ejecutar el comando siguiente:

```
$ kubeadm init
```

Esto creará la configuración para los componentes del plano de control y *kubelet*. *kubeadm* genera un *token* a la salida de ese comando. Ese token deberá usarse en el siguiente paso.

Esta operación también ha generado las credenciales de administrador del clúster. Estas se encuentran en la ruta `/etc/kubernetes/admin.conf`. Es común conectarse desde otro equipo

[79] https://coreos.com/etcd

para operar. Para habilitar esa opción basta con copiar el archivo *admin.conf* a la ruta *~/.kube/config* del equipo desde el que se desee interactuar con la API de *Kubernetes*.

4.3.3.d. Creación de los demás nodos

Una vez el nodo *master* está activo, es el momento de unir el resto de nodos. *kubeadm* también se encarga de esto. Para ello, se ejecuta el siguiente comando en cada uno de los nodos, donde *token* es el *token* obtenido en el paso anterior:

```
$ kubeadm join --token <token> 192.168.15.8
```

Se puede comprobar que estas operaciones han tenido éxito listando los nodos del clúster con el siguiente comando:

```
$ kubeclt get nodes
NAME      STATUS    ROLES     AGE       VERSION
moleman0   Ready     master    1d        v1.10.2
moleman1   Ready     master    3h        v1.10.2
moleman2   Ready     master    3h        v1.10.2
```

4.3.3.e. Aislamiento del *master*

Por defecto, *kubeadm* configura *Kubernetes* de forma que ningún *pod* puede ser ejecutado en el nodo *master*. Esto es adecuado en entornos de producción reales por motivos de seguridad.

En el caso de este proyecto se prefiere poder usar los recursos del *master* para ejecutar carga útil. El siguiente comando permite habilitar esa posibilidad:

```
$ kubectl taint nodes --all node-role.kubernetes.io/master-
node "moleman0" untainted
taint key="dedicated" and effect="" not found.
taint key="dedicated" and effect="" not found.
```

4.3.4. Instalación de la capa de red de *Kubernetes*

Kubernetes necesita una capa de red lógica que permita la comunicación entre los containers. Existe un estándar para este tipo de subsistemas llamados *CNI*[80]. Un *CNI* se encarga de facilitar la comunicación entre un container y otro que corren en diferentes nodos del clúster.

Existen varios *CNI open-source* populares: *calico*[81], *kube-router*[82], *flannel*[83], *weave*[84].

En primer lugar se ha intentado usar *kube-router*, puesto que por su implementación con componentes de más bajo nivel promete un rendimiento mejor. En las primeras pruebas apareció un problema: los servicios no hacían la redirección a los *pods* correspondientes. Tras una investigación, se descubrió que *kube-router* no soporta *named-ports*[85]: puertos que son referenciados por un nombre en lugar de por el valor directamente.

Tras descubrirse la limitación anterior, se ha optado por usar *weave*. La instalación de *weave* es bastante sencilla una vez se tiene un clúster de *Kubernetes* operativo[86]. Basta con aplicar el siguiente manifiesto:

```
$ kubectl apply -f
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64
| tr -d '\n')"
```

4.3.5. Soporte para *Kubernetes LoadBalancers*

Un recurso de tipo *LoadBalancer* en *Kubernetes* es una abstracción que configura un balanceador de carga cuya implementación varía en función de la infraestructura que soporte el clúster.

Los mayores proveedores de cloud (*AWS, Google Cloud* y *Microsoft Azure*) permiten crear balanceadores de carga. Cuando *Kubernetes* funciona sobre uno de esos entornos, los recursos de

[80] *Container Network Interface*
[81] https://github.com/projectcalico/cni-plugin
[82] https://github.com/cloudnativelabs/kube-router
[83] https://github.com/coreos/flannel
[84] https://github.com/weaveworks/weave
[85] https://github.com/cloudnativelabs/kube-router/issues/338
[86] https://www.weave.works/docs/net/latest/kubernetes/kube-addon

tipo *LoadBalancer* interactúan con la API del cloud provider para instanciar y configurar un balanceador.

Kubernetes no tiene una implementación para balanceadores de carga en clústeres *bare metal*. Muy recientemente (a finales de 2017), *Google* ha liberado un proyecto llamado *MetalLB*[87] que soluciona este problema.

MetalLB tiene dos modos de funcionamiento. El primer modo hace que *Kubernetes* se integre con equipamiento de red específico mediante protocolo *BGP*[88]. Este modo no es conveniente para este proyecto, puesto que no se dispone de equipamiento de red con soporte para *BGP*.

El segundo modo, llamado modo *Layer 2*, no requiere de hardware específico puesto que basta con que un proceso corriendo en *Kubernetes* tenga acceso directo a la capa de red de los nodos del clúster. Por tanto, este será el modo a usar.

El modo *Layer 2* funciona de la siguiente manera. Se reserva un rango de IPs en la red: en este caso se ha configurado el router para que no asigne las direcciones entre `192.168.15.100` y `192.168.15.200` (ver apartado Hardware y configuración de red). *MetalLB* crea un *daemonSet* en *Kubernetes*. Esto hace que en cada nodo del clúster se ejecute un *pod* (*speaker*) con acceso a la capa de red de la máquina. También se ejecuta otro *pod* llamado *controller* del que sólo hay una instancia en todo el clúster (a no ser que se configuren réplicas). Cuando se crea un servicio de tipo *LoadBalancer*, el *controller* hace que se le asigne una de las IPs reservadas. En ese momento *MetalLB*, a través de los *speakers*, usa el protocolo *ARP*[89] para hacer que la dirección IP asignada se resuelva a la dirección física de uno de los nodos. Esto es posible ya que los container de los *speakers* están configurados para usar directamente la interfaz de red del host, por lo que pueden interactuar a nivel 2 en la *LAN*.

Para el caso de este proyecto, se ha instalado *MetalLB* en el clúster con el siguiente comando:

```
$ kubectl apply -f
https://raw.githubusercontent.com/google/metallb/v0.6.2/manifests/metallb.
yaml
```

[87] https://github.com/google/metallb
[88] *Border Gateway Protocol*: protocolo que permite a intercambiar información de enrutado entre *AS* (*Autonomous Systems*). Estos *AS* son componentes de la red con inteligencia para decidir sobre el enrutado de paquetes en base a reglas preestablecidas.
[89] *Address Resolution Protocol:* revuelve cual es la dirección física MAC correspondiente a una dirección IP dentro de una red local.

Los recursos no se crearán hasta que se proporcione una configuración apropiada con el rango de direcciones reservado. Para ello se ha aplicado:

```
$ cat <<EOF | kubectl create -f -
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: my-ip-space
      protocol: layer2
      addresses:
      - 192.168.15.100-192.168.1.200
EOF
```

Se puede comprobar que la instalación ha tenido éxito observando el estado de los pods en el namespace propio que crea *MetalLB*:

```
$ kubectl -n metallb-system get pods
controller-10d9054faa-h5to3   1/1       Running   0          11m
speaker-3xj5i                 1/1       Running   0          10m
speaker-de3af                 1/1       Running   0          11m
speaker-a0mbo                 1/1       Running   0          11m
```

4.3.6. Soporte para *Helm Charts*: instalación de *tiller*

Para el desarrollo del proyecto se va a necesitar soporte para *Helm*[90], el *package manager* de *Kubernetes*. *Helm* se compone de dos partes: un cliente de línea de comandos y un servidor en el clúster. La parte servidor se denomina *tiller*.

Tiller hace de interfaz entre la API de *Kubernetes* y el cliente. Se compone de un recurso tipo *service* y uno tipo *deployment*. Por defecto, *tiller* se instala en el *namespace* de sistema de *Kubernetes*

[90] https://github.com/kubernetes/helm

(*kube-system*), teniendo así privilegios para crear y eliminar recursos. Es posible hacer otras configuraciones más restrictivas, pero puesto que para este proyecto se considera suficiente la configuración por defecto. Para instalar *tiller* basta ejecutar el siguiente comando una vez se tiene *helm* instalado y credenciales de administrador en el clúster:

```
$ helm init
```

Para comprobar si se ha instalado correctamente se puede ejecutar *helm* con el verbo *version*, y debería mostrar la versión instalada en el servidor:

```
$ helm version
Client: &version.Version{SemVer:"v2.9.1",
GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710",
GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.9.1",
GitCommit:"20adb27c7c5868466912eebdf6664e7390ebe710",
GitTreeState:"clean"}
```

4.3.7. Soporte para *Kubernetes Ingress: IngressController*

Para que la abstracción de los *Ingress* esté disponible, el clúster necesita contar con un *IngressController*.

Para una instalación *bare metal* los *IngressControllers open-source* más populares son los siguientes:

Contour[91]

Contour está desarrollado por *Heptio*[92] una *startup* que ofrece soporte de *Kubernetes*.

Utiliza el popular *envoy*[93] como proxy inverso y balanceador de carga. *Envoy* es conocido por ser realmente rápido y eficiente. Está desarrollado por *lyft*, una compañía de transportes con conductor.

Istio Ingress[94]

[91] https://github.com/heptio/contour
[92] https://heptio.com
[93] https://www.envoyproxy.io
[94] https://istio.io/docs/tasks/traffic-management/ingress.html, https://github.com/istio/istio

Istio ofrece más funcionalidad que la de un simple *proxy* inverso. Proporciona una completa plataforma para gestionar y monitorizar microservicios. Al igual que *contour,* también usa *envoy*.

Nginx Ingress Controller[95]

Esta es la solución proporcionada por la comunidad de *Kubernetes*. Consiste en una instancia de *nginx*[96]*,* un potente servidor web muy usado como balanceador de carga, fachada para microservicios y *API gateway*.

El proyecto se ha hecho bastante popular, y parece ser la elección *de facto*. Existen además múltiples proyectos que se integran con este para añadir funcionalidades.

Por popularidad y capacidad de expansión de funcionalidad, se va a optar por *Nginx Ingress Controller* para este proyecto.

El proyecto ofrece varios métodos de instalación. Uno de ellos está basado en un *Helm Chart*[97], lo que resulta muy conveniente en este caso, ya que *Helm* está ya habilitado en el clúster del proyecto.

Se instala el controller con el siguiente comando:

```
$ helm install stable/nginx-ingress --set rbac.create=true --name
ingress-nginx
```

Es necesario habilitar la opción `rbac.create` puesto que el clúster de este proyecto tiene tiene control de acceso basado en *RBAC*[98] (activado por defecto con *kubeadm*).

Se puede verificar que la instalación ha sido exitosa mostrando el estado de los dos *pods* que componen *Nginx Ingress Controller* con el siguiente comando:

```
$ kubectl -n ingress-nginx get pods
NAME                                    READY     STATUS    RESTARTS   AGE
default-http-backend-5c6d95c48-rbrwp    1/1       Running   3          2d
```

[95] https://kubernetes.github.io/ingress-nginx, https://github.com/kubernetes/ingress-nginx
[96] https://www.nginx.com
[97] https://kubernetes.github.io/ingress-nginx/deploy/#baremetal
[98] *Role Based Access Control*

```
nginx-ingress-controller-7677864cc9   1/1       Running   0          2d
```

Automáticamente, se crean también dos servicios:

```
$ kubectl -n ingress-nginx get service
NAME                 TYPE      clúster-IP    EXTERNAL-IP  PORT(S)
default-http-backend clusterIP 10.99.115.134 <none>       80/TCP
ingress-nginx        NodePort  10.105.75.244 192.168.15.8 80:30522/TCP,
                                                          443:31336/TCP
```

El primero de ellos, *default-http-backend*, es un servicio web que devuelve siempre una página con *404 Not Found*. Será a donde *Nginx* redirija todas las peticiones que no tengan un *Ingress* correctamente configurado.

El segundo de ellos es el servicio que está expuesto al exterior. Se puede observar como está escuchando los puertos 80 (*HTTP*) y 443 (*HTTPS*). Este servicio dirigirá el tráfico al servicio configurado en el recurso *Ingress* correspondiente según la ruta.

4.3.8. Soporte para TLS en *Ingress*: *CertManager* y certificados con *Let's Encrypt*

Por defecto, *Nginx Ingress Controller* habilita HTTPS, pero lo hace con un certificado auto firmado.

Es posible instalar una extensión que crea certificados válidos para cada ingress. Los certificados usan la tecnología de *Let's Encrypt*[99]. *Let's Encrypt* es una autoridad certificadora gratuita, automática y *open-source*.

Para que los certificados se generen y configuren automáticamente, se usa la extensión *cert-manager*[100]. *Cert-manager* se puede instalar con *helm* de la siguiente forma:

```
$ helm install --name cert-manager --namespace kube-system
stable/cert-manager
```

[99] https://letsencrypt.org
[100] https://github.com/jetstack/cert-manager, https://cert-manager.readthedocs.io

Se puede comprobar el estado de esta forma:

```
$ helm status cert-manager
```

Devolverá una lista de todos los recursos creados y de su estado.

4.3.8.1. Configurar *Issuers*

Para poder generar certificados es necesario crear un recurso de tipo *Issuer*. Esto no es más que generar una autoridad certificadora propia que permita firmar certificados. Esta es la operación realizada en el clúster:

```
$ cat <<EOF | kubectl create -f -
apiVersion: certmanager.k8s.io/v1alpha1
kind: clústerIssuer
metadata:
  name: letsencrypt-prod
  namespace: kube-system
spec:
  acme:
    # The ACME server URL
    server: https://acme-v01.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: <email>
    # Name of a secret used to store the ACME account private key
    privateKeySecretRef:
      name: letsncrypt-prod
    # Enable HTTP01 validations
    http01: {}
EOF
```

La anterior operación da como resultado la creación de un *clusterIssuer* con nombre `letsencrypt-prod` que podrá ser referenciado en un *Ingress* para crear certificados a demanda.

4.3.9. Instalación de *Kubernetes Dashboard*

Para la facilitar la interacción con *Kubernetes*, es posible instalar un panel de control web[101].

[101] https://github.com/kubernetes/dashboard

Para instalarlo basta con ejecutar:

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

Una vez finalizado, se puede acceder al panel de control haciendo un proxy de la siguiente manera:

```
$ kubectl proxy
```

Y accediendo a http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/.

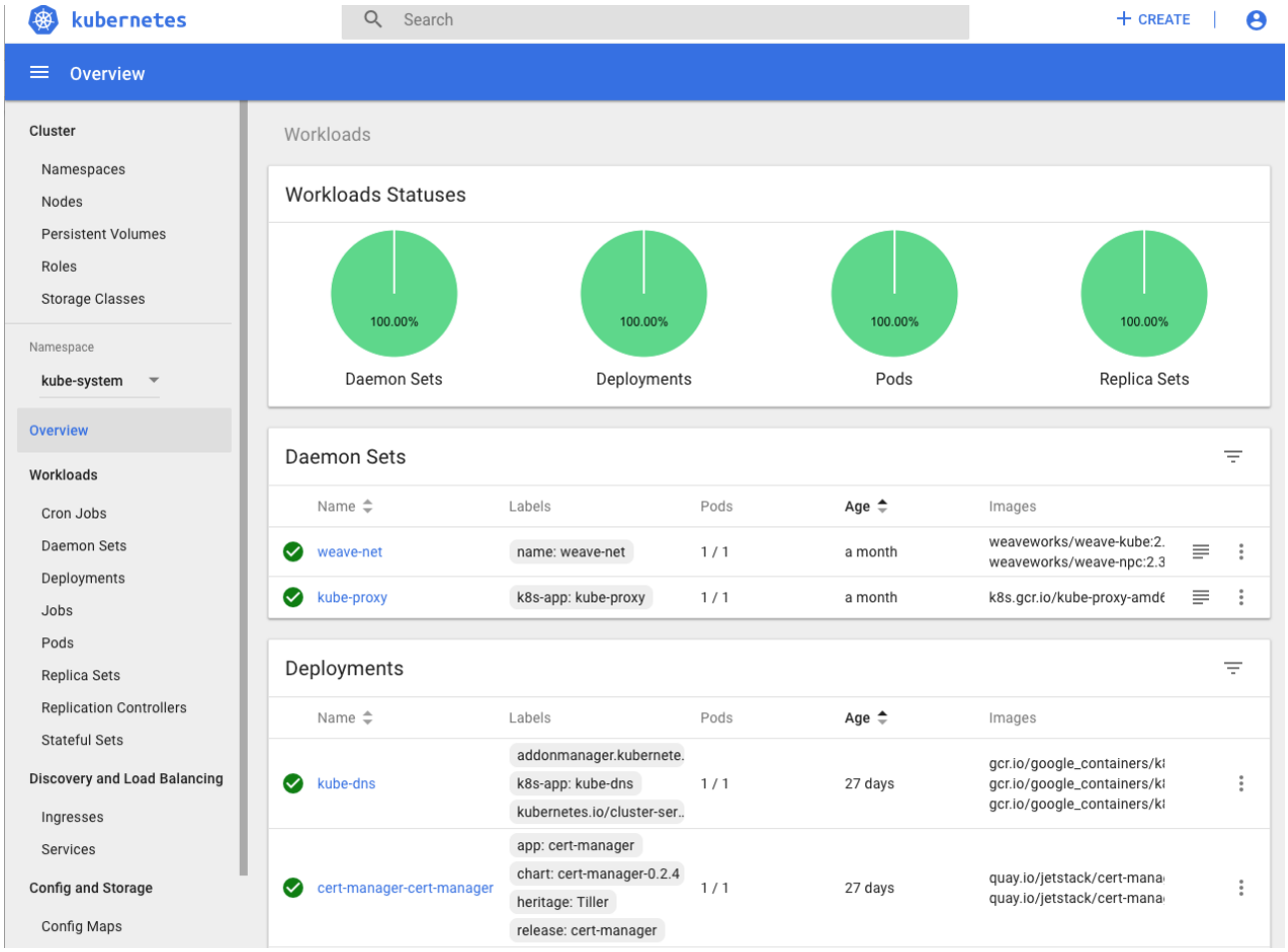


Fig 20. *Dashboard* de *Kubernetes*.

4.4. Arquitectura de la solución

La solución consta de un colector de datos, una base de datos, y un panel de visualización.

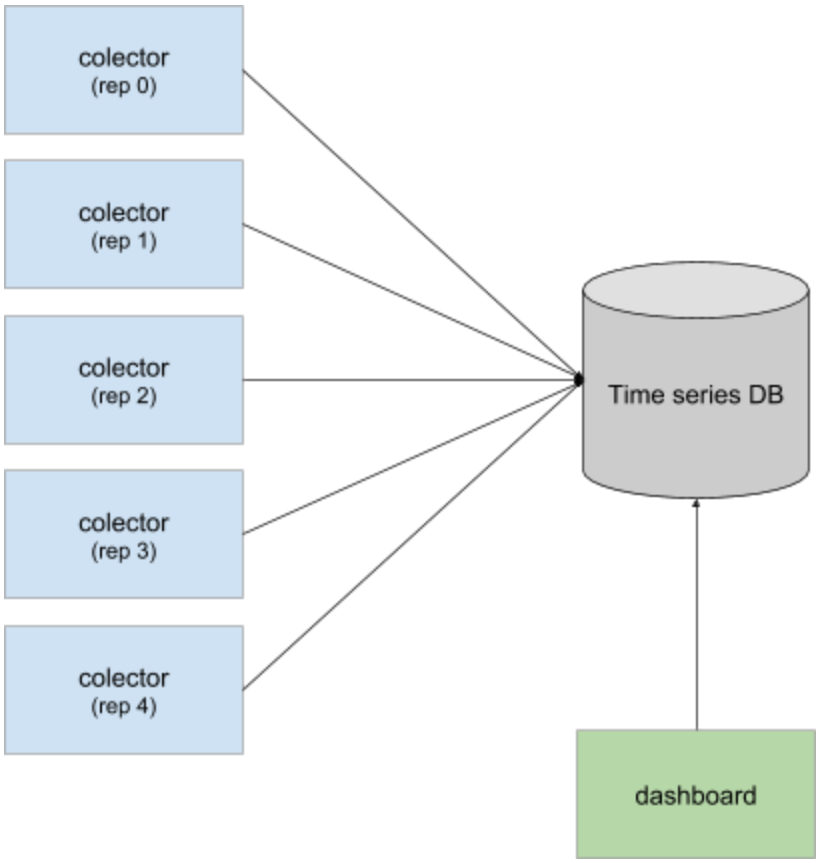


Fig 21. Esquema básico de arquitectura de la solución.

El colector expone a internet una interfaz CoAP, punto de entrada de los datos provenientes de dispositivos IoT. Puede haber múltiples réplicas del colector sobre las que balancear la carga. Los colectores procesan los mensajes CoAP, extrayendo la información útil enviada por el dispositivo IoT y dándole el formato adecuado para su inyección en la base de datos. Esta inyección se realiza antes de enviar el mensaje de respuesta al dispositivo IoT, de forma que en la respuesta se indica si la operación ha tenido éxito o no.

La base de datos recibe y almacena la información recibida del colector. Puesto que la naturaleza de los datos son medidas de dispositivos IoT, lo más natural es almacenar esa información en forma de series temporales.

El panel de visualización se conecta a la base de datos y permite observar las medidas y visualizar gráficos.

4.5. Elección de la base de datos

Bajo la consigna de usar una base de datos open-source, que pueda escalar con topologías de alta disponibilidad y que tenga un buen soporte para series temporales, se hace esta comparación de motores de bases de datos.

Prometheus[102]

Es un proyecto open-source con todavía corto recorrido (unos dos años). Es un sistema de recogida de eventos implementado en *Golang*.

Prometheus permite hacer multitud de operaciones entre los eventos recogidos: medias, cálculos de incrementos, histogramas, cuartiles, percentiles y muchos otros.

Es un sistema robusto, pero muy simple. Apenar hay tres tipos de datos que puede tratar.

Prometheus se integra muy bien con Grafana. Grafana es una herramienta web para crear *dashboards* de visualización de datos.

La mayor limitación de Prometheus en el contexto de este proyecto es su modelo de recogida de datos. No es posible enviar datos directamente a Prometheus, sino que Prometheus se configura para que recolecte los datos de forma periódica en un punto a través de peticiones HTTP, por ejemplo. Puesto que los colectores de datos van a ser efímeros, sería muy probable que se perdieran métricas entre escaneo y escaneo.

InfluxDB[103]

InfluxDB es una de las bases de datos orientadas a series temporales que más rápido está adoptando usuarios. Tiene una gran comunidad detrás apoyando el desarrollo. Está implementado en *Golang*.

[102] https://prometheus.io, https://github.com/prometheus
[103] https://www.influxdata.com, https://github.com/influxdata/influxdb

El rendimiento de InfluxDB es muy bueno. Con un único nodo, es capaz de ingerir alrededor de 350 mil métricas por segundo.

Además las *queries* también se resuelven con bastante rapidez.

Se puede encontrar en internet bastante material de cómo desplegar InfluxDB sobre containers. No hay tanto material sobre cómo desplegarlo sobre Kubernetes específicamente.

Druid[104]

Druid es un proyecto implementado en *Java* que hace uso de *HDFS* para ofrecer un almacén de datos orientado a columnas.

Se puede ver Druid como un "todo en uno", pues aporta también toda una capa de visualización de datos.

Ingesta datos a un ritmo relativamente rápido: 20 mil métricas por segundo con un solo nodo. Sin embargo las *queries* tardan bastante (en comparación con otras tecnologías) en ser procesadas.

La documentación también es bastante rica, pero no tiene una comunidad tan grande como otros proyectos aquí presentados.

Elasticsearch[105]

Elasticsearch es un motor de búsqueda y análisis de datos implementado en *Java*.

Es capaz de indexar millones de datos y realizar búsquedas y operaciones sobre los mismos de forma muy eficiente.

Es capaz de ingerir unas 50 mil métricas por segundo con un solo nodo. También procesa *queries* de forma rápida.

Elasticsearch almacena la información en forma de documentos. Se puede configurar como se indexa cada campo del documento. Internamente, cada evento recibido se asocia con una huella

[104] http://druid.io, https://github.com/druid-io/druid
[105] https://www.elastic.co/products/elasticsearch, https://github.com/elastic/elasticsearch

temporal. Combinando estas dos propiedades, se puede usar Elasticsearch para tratar series temporales.

Dentro de la misma suite de productos está Kibana. Kibana es una herramienta web para configurar Elasticsearch, visualizar datos y hacer gráficas. Se integra a la perfección con Elasticsearch.

En internet hay muchísimo buen materias sobre Elasticsearch. Es un proyecto muy usado y con una gran comunidad detrás.

MongoDB[106]

MongoDB es una base de datos orientada a documentos. Está implementada en C++.

Es de propósito general, pero su punto fuerte es que es capaz de almacenar e indexar grandes cantidades de datos.

Dicho eso, no es fácil configurar MongoDB para que tenga pueda aceptar datos a una velocidad comparable a otros proyectos ya comentados. Sí que tiende a superar a los demás cuando se trata de documentos grandes en lugar de pequeños eventos.

MongoDB también puede funcionar como base de datos de series temporales, pues gestiona internamente las huellas temporales de las acciones.

Es la base de datos de documentos *de facto*, y esa gran aceptación hace que haya disponible muchísima información y soporte online.

Su funcionamiento sobre Docker y Kubernetes está más que probado.

Tras evaluar las diferentes opciones, hay dos que, *a priori*, destacan sobre las demás: *InfluxDB* y *Elasticsearch*. Finalmente, se opta por *Elasticsearch*, valorando muy positivamente que *Kibana* puede solventar gran parte de los requisitos de visualización de datos que hay en este proyecto.

[106] https://www.mongodb.com, https://github.com/mongodb/mongo

4.6. Topología real referente a la tecnología de orquestación escogida

4.6.1. Topología de la solución en Kubernetes

En este apartado se describe el diseño de la solución a nivel de Kubernetes. Se justifica cada uno de los recursos declarados, así como su configuración y relación con el resto.

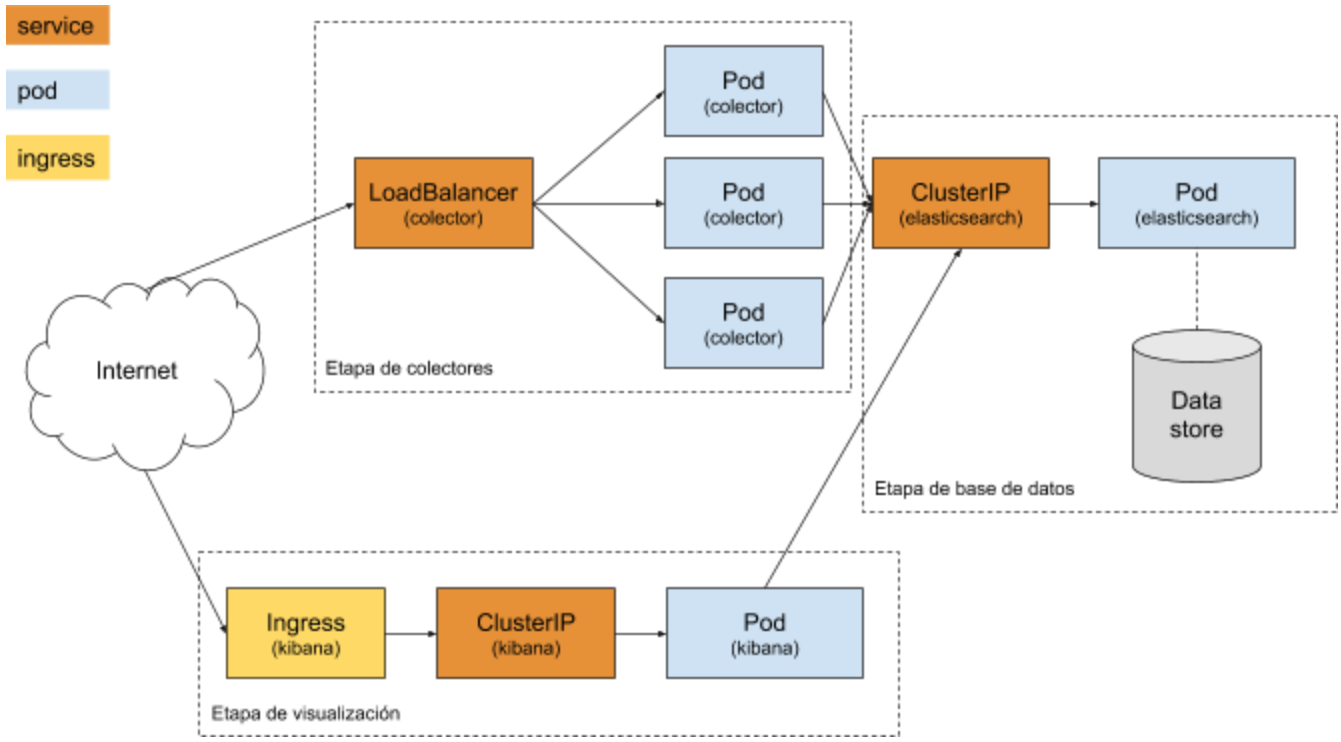


Fig 22. Topología de la solución en base a componentes de *Kubernetes*.

Se diferencian tres grandes bloques o etapas que están marcados en el diagrama con línea discontínua.

4.6.1.a. Etapa de colectores

El bloque relativo al colector de datos cuenta con un servicio de tipo *LoadBalancer* que está expuesto a internet. Este es el punto de entrada de todos los eventos provenientes de los dispositivos IoT. Este servicio escucha en un puerto UDP. Puesto que el protocolo de nivel de aplicación escogido es CoAP se usará el puerto por defecto para ese protocolo, 5683. Este servicio

usa un selector para escoger que el destino del tráfico entrante sean los *pods* que tienen el proceso colector corriendo.

Idealmente, se habría optado por poner un *ingress* delante del servicio (como se ha hecho un la etapa de visualización), pero actualmente *ingress* no tiene soporte para un protocolo diferente de HTTP, por lo que esta opción queda descartada. Puesto que esta etapa de colectores usa un puerto diferente de la de Kibana, esto no es mayor problema. Es caso de que fueran dos servicios web escuchando por el mismo puerto, *ingress* sería la única opción.

Como se aprecia en el esquema, existen varias réplicas del *pod* que corre el proceso colector de datos. Esto es posible ya que esa parte de la aplicación se ha diseñado a conciencia para que sea fácilmente escalable de forma horizontal. Realmente estos *pods* se crean a partir de un recurso de tipo *deployment* que es quien realmente define qué son esos *pods*, así el número de réplicas.

Los recursos que puede consumir cada *pod* están también limitados por diseño. En el manifiesto del *deployment* correspondiente se especifican unos límites de CPU y memoria que el proceso no puede traspasar. Si el proceso los traspasara, por una fuga de memoria por ejemplo, Kubernetes se encargaría de eliminar ese *pod* y arrancar otro nuevo.

Los procesos colectores de datos se conectan al servicio *elasticsearch* definido en la siguiente etapa.

4.6.1.b. Etapa de base de datos

Esta etapa no está expuesta al exterior. Se expone al resto del clúster a través de un servicio de tipo *clusterIP* que le asigna una IP privada interna.

Este servicio redirige el tráfico entrante hasta un único *pod* (también definido en un recurso de tipo *deployment*) que ejecuta el proceso del motor de la base de datos.

El *pod* monta un volúmen de datos externo, que es el único punto de todo el sistema en el que se persiste el estado.

Kubernetes soporta múltiples backends de almacenamiento. Elegir uno u otro depende de la infraestructura real que haya por debajo. Para este proyecto se está usando un clúster *bare metal* y se supondrá que no hay disponible ningún servicio de tipo *NFS*[107] o similar. Por tanto se ha optado

[107] *Network File System*. Es un protocolo que permite ofrecer un sistema de archivos compartido en una red.

por por la implementación más genérica, los ya mencionados *persistentVolumes*. Para que esta configuración sea posible, también se crea el correspondiente *persistentVolumeClaim*.

Elasticsearch es una aplicación que demanda muchos recursos. Establecer ese límite en el manifiesto del *deployment* de esta etapa es un compromiso delicado. Si se establece un límite muy bajo, Kubernetes podría matar el proceso de Elasticsearch demasiado a menudo. Si se deja un límite demasiado alto, el proceso podría crecer en memoria descontroladamente. La gestión de memoria de *Java* está muy lejos de ser óptima.

4.6.1.c. Etapa de visualización

Esta etapa, como la de colectores, está expuesta al exterior. A diferencia de la primera, esta etapa usa un *ingress* en lugar de exponer directamente el servicio. Esto permite configurar de forma sencilla un dominio para la aplicación, así como una terminación *TLS* con un certificado válido de forma que Kibana sea accedido mediante *HTTPS*.

El tráfico que entra en el *ingress* es enrutado hacia el servicio de tipo *clusterIP* que hay a continuación. Este servicio no es más que la puerta de entrada al *pod* que corre el proceso de Kibana.

Kibana está configurado para conectarse al servicio de Elasticsearch, de donde se nutre de los datos recogidos por los colectores para mostrar las gráficas.

Todos los pods mencionados toman su configuración mediante *configMaps* que son creados como parte de los manifiestos de despliegue.

4.7. Diseño e implementación del software del colector

4.7.1. Runtime

Dado que van a ejecutarse múltiples réplicas del proceso en paralelo, lo ideal sería usar un *runtime* que fuera óptimo para funcionar en un sólo hilo. Son muchas las opciones disponibles, y al final se reduce a una elección algo arbitraria. Cada runtime tiene su ecosistema de librerías.

Se ha decidido optar por *node.js*[108], puesto que existen librerías de *Javascript* para manejo de mensajes CoAP, así como otras utilidades que se prevé se necesitarán.

4.7.2. Gestor de dependencias

Dentro del ecosistema de *node.js*, es común el uso de gestores de paquetes o gestores de dependencias. La opción nativa, y la más común, es *npm*[109]. Sin embargo, se ha optado por usar *yarn*[110]. *Yarn* es un proyecto creado por *Facebook* para gestionar dependencias en proyectos *Javascript*. Es particularmente interesante porque tiene un algoritmo de resolución de dependencias cruzadas que mucho más óptimo y rápido que *npm*. Esto se traduce en que las dependencias tardan menos en instalarse y que en que el resultado ocupa menos espacio en disco. Esto lo consigue reduciendo el número de versiones de una librería a instalar buscando versiones compatibles entre las dependencias recursivas.

Se definen dos sets de dependencias: uno para producción y otro para desarrollo. El primero es un subconjunto del segundo. Durante el desarrollo se requiere de librerías que no son necesarias en producción, como es el caso de las librerías relativas a *linters* de código y tests.

4.7.3. Programa principal

Al ejecutar el programa este lee la configuración de desde disco y la carga en el entorno.

A continuación comprueba la conectividad con la base de datos. En caso de no tener conectividad, el proceso finaliza, dejando así al orquestador (Kubernetes en este caso) gestionar el reinicio de la misma.

Una vez se valida la conectividad con la base de datos, se cargan el sistema de *middlewares* y se entra en un bucle infinito que está siempre leyendo mensajes CoAP en el puerto configurado.

Por cada petición se imprime cierta información relevante en la salida estándar. Dicha salida es asumida por Kubernetes, y puede consultarse a través de su API y del dashboard de Kubernetes.

[108] https://nodejs.org
[109] https://www.npmjs.com
[110] https://yarnpkg.com

4.7.4. Middlewares

Los *middlewares* son las diferentes capas por las que pasa capa mensaje CoAP recibido.

El primer *middleware* se asegura que de el mensaje tiene un formato adecuado, lo cual para esta aplicación significa que contenga una cabecera que diga que está codificado en JSON.

El segundo *middleware* configura la futura respuesta con algunas cabeceras por defecto.

El tercer *middleware* es un *router*. Esta capa se encarga de leer la URL del mensaje y encaminarla hasta el *controller* adecuado para esa ruta.

4.7.5. Rutas

Se definen dos rutas diferentes. Una de ellas es una ruta de comprobación del sistema o *healthcheck*. Una petición de tipo `GET` a esta ruta (`/health`) devolvería el estado del sistema. Esto es útil para los usuarios, pero sobre todo para el correcto funcionamiento de Kubernetes. El *pod* de los colectores está configurado con una *readinessProbe* que está constantemente consultando este *endpoint*. El *LoadBalancer* sólo dirigirá tráfico hasta un *pod* colector cuando la *readinessProbe* haya devuelto un resultado positivo.

La otra ruta disponible es la ruta que recoge los eventos. Esta es la ruta a la que los dispositivos IoT envían sus eventos. Cualquier petición de tipo `POST` a `/events` será recibida en el *controller* de eventos. Este *controller* hará comprobaciones de formato adicionales y en caso de que el cuerpo del mensaje cumpla con el formato esperado, recodificará la información y la enviará a la base de datos.

4.7.6. Control de errores

Durante el procesado de las peticiones puede surgir algún error. Todos estos errores son capturados y tratados adecuadamente de forma que la respuesta que se mande al usuario de vuelta sea consistente con el estándar CoAP.

En el caso de que el error sea debido a que la petición original fuera incorrecta por algún motivo, se enviará un mensaje de respuesta código del tipo `'4.xx'`, así como una descripción del error.

En caso de que el error sea debido a un error interno, el mensaje de respuesta contendrá un código `'5.xx'` y también una descripción del error.

4.7.7. Tests

Para asegurar dentro de lo posible que al hacer cambios al código no se altera el funcionamiento esperado de la aplicación, se ha añadido una suite de tests.

Para ejecutar estos tests se usa *mocha*[111].

Estos tests se ejecutan dentro del CI.

4.7.8. Linter

Un linter es una utilidad que se asegura que el código cumple ciertas normas.

Para este proyecto se ha usado el linter para *Javascript* llamado *eslint*[112]. Se ha elegido la guía de estilo y normal de *airbnb*[113], por ser una de las más populares entre la comunidad *Javascript*.

Durante CI se fuera a que el código cumpla todas las normas del linter. Esto asegura que el estilo sea consistente en todo el proyecto y también previene algunos errores comunes.

4.8. Diseño e implementación del pipeline

4.8.1. Sistema de CI/CD en el propio clúster

El secreto de un ciclo de desarrollo rápido es tener un buen sistema de CI/CD. Como se vió al principio de este documento, en entornos complejos como Kubernetes en los que hay que construir imágenes, testearlas y desplegarlas esto pasa de ser una recomendación a ser prácticamente imprescindible.

Puesto que ya se tiene un clúster de Kubernetes funcional, se ha decidido aprovechar el mismo para soportar el propio sistema de CI/CD.

[111] https://mochajs.org
[112] https://eslint.org
[113] https://github.com/airbnb/javascript

Se ha optado por usar *Gitlab*[114], ya que aúna en un sólo producto las funcionalidades de repositorio de código, y CI/CD (además de muchas otras que no son relevantes para este caso).

Actualmente, el equipo detrás de *Gitlab* está desarrollando una nueva versión de *Gitlab* totalmente nativa en Kubernetes. En el momento de desarrollar este proyecto, no hay disponible ninguna versión 100% funcional en Kubernetes, por lo que se han implementado los manifiestos para ejecutar *Gitlab* en Kubernetes *ex professo* para la ocasión.

En el siguiente esquema se representa la topología implementada para ejecutar *Gitlab* sobre Kubernetes.

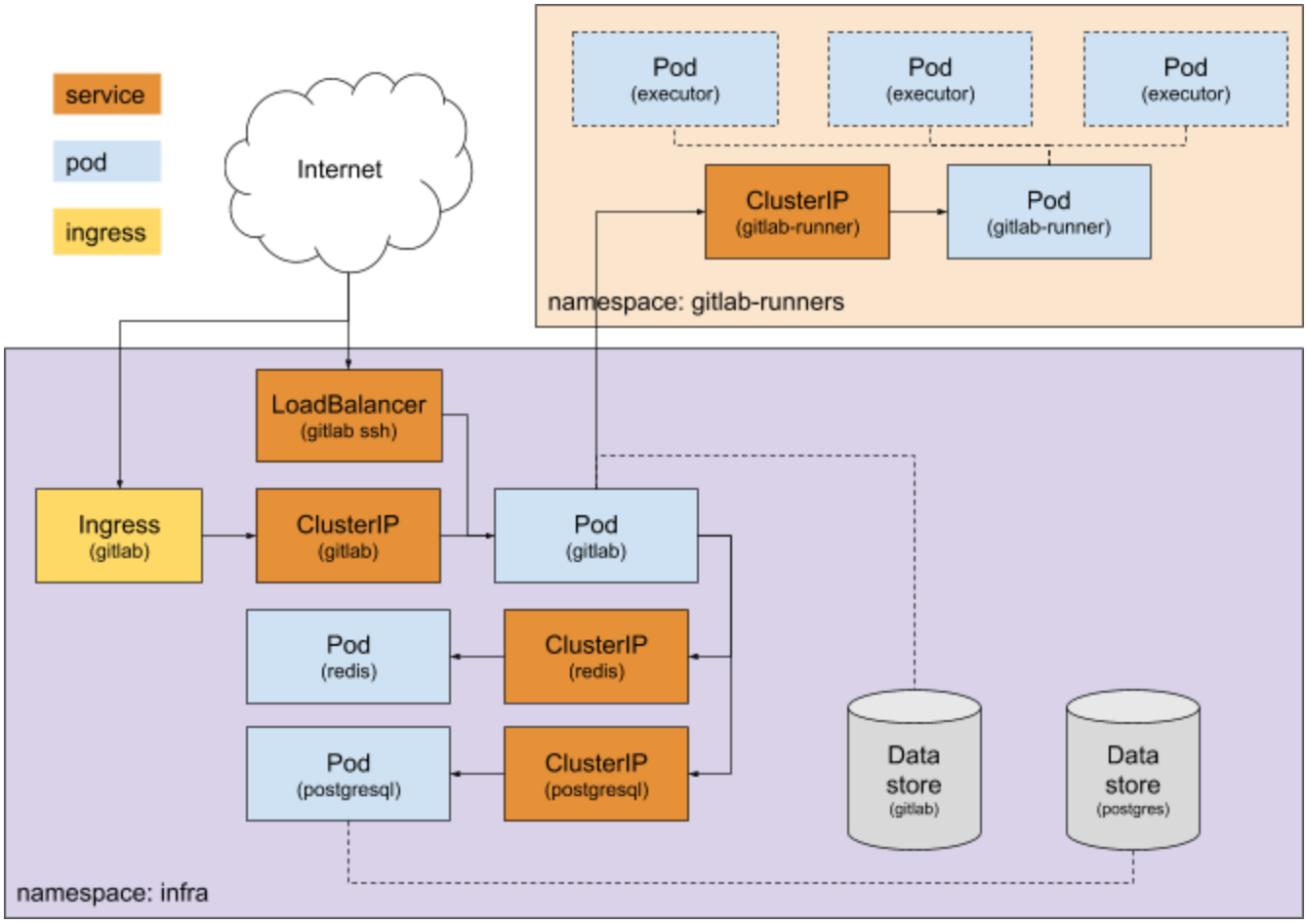


Fig 23. Topología del *pipeline* en base a componentes de *Kubernetes*.

En primer lugar, cabe explicar que hay dos *namespaces* diferenciados. Esto permite que toda la parte de infraestructura de *Gitlab* esté aislada de los ejecutores del CI. Este aislamiento es delicado, ya que

[114] https://about.gitlab.com/

esos ejecutores van a correr código arbitrario descargado de internet (por ejemplo cuando se descargan e instalan las librerías del colector mientras se construye su imagen). El *namespace* de *infra* tiene ciertos privilegios que no son deseables que caigan en manos de software malicioso.

El acceso de desde el exterior a la web va a traves de un *ingress*. Este *ingress* dirige el tráfico a un servicio *clusterIP* que lo enruta hacia el *pod* que corre los procesos principales de *Gitlab*. Existe otro servicio de tipo *LoadBalancer* que es por donde entra el tráfico SSH hasta el *pod* de *Gitlab*. SSH es usado por el cliente *git* para interactuar con los repositorios. El *pod* de *Gitlab* monta un *persistentVolume* que usa para persistir los repositorios.

Gitlab se conecta a un servidor *redis*[115] para guardar información de sesiones. También lo usa como caché.

Gitlab también se conecta a una base datos *Postgres*[116] donde guarda parte de su estado. El *pod* de *Postgres* monta un *persistentVolume* en el que almacena los datos.

Cuando *Gitlab* tiene que ejecutar operaciones relativas al CI/CD como por ejemplo construir imágenes docker o ejecutar los tests del código, usa un *executor*. Un *executor* es un nuevo pod aislado que se crea en Kubernetes (en el namespace *gitlab-runners*) y que ejecuta en un entorno controlado las operaciones que se le ordenan. Estos *executors* son creados por *gitlab-runner* bajo demanda. Son creados a partir de una imagen base.

Dadas las peculiaridades de este proyecto, esta imagen base se ha tenido que crear específicamente (ver An.2 Dockerfile para imagen base de Gitlab runner executors). La imagen debía tener instalado el cliente de Kubernetes, puesto que debe hablar con la API de Kubernetes para desplegar el proyecto. También debía contar con capacidad de ejecutar docker, requisito necesario tanto para construir imágenes como para testearlas.

El hecho de ejecutar docker dentro de un container complica bastante la situación. Actualmente gran parte de la comunidad está buscando una buena solución a este problema, pero a día de hoy sólo hay dos soluciones a medias. La primera de ellas es ejecutar el container en modo privilegiado y ejecutar docker dentro. Esto es tremendamente inseguro, ya que compromete a la máquina *host* al dar privilegios totales a algo que desde el principio estaba aislado por diseño por ser inseguro. La segunda solución es compartir el socket de docker de la máquina host. De esta forma el cliente

[115] https://redis.io. Redis es una base de datos en memoria.
[116] https://www.postgresql.org. Postgres es una base de datos SQL.

docker de dentro del container puede hablar con el demonio docker del host y ejecutar containers en el. Esta solución tampoco es ideal, ya que igualmente todos los containers que se estén ejecutando en el host quedan expuestos.

Para este proyecto se ha optado por la segunda solución pero con una variante que mitiga sus problemas de forma considerable. La máquina host está ejecutando realmente dos demonios docker, configurados para no solaparse. Uno de ellos es el que ve el plano de control de Kubernetes, y el otro está dedicado para los *executors*. El script que inicia y configura este segundo demonio *docker* está en An.3 Script de inicio de demonio docker secundario.

4.8.2. Empaquetado y construcción de las imágenes

4.8.2.a. Dockerfiles

Las imágenes docker que se despliegan como parte de los *pods* se generan mediante *Dockerfiles*.

Un *Dockerfile* es un archivo de instrucciones que interpretadas una tras otra da lugar a una imagen docker.

Cada sentencia del *Dockerfile* se convertirá en una capa de la imagen por lo que, para minimizar el tamaño, cobra importancia hacer dentro de la misma sentencia operaciones que involucren movimiento de muchos datos de los que luego se va a borrar gran parte.

En este proyecto hay *Dockerfiles* para construir las imágenes base de los *executors* de *Gitlab* y para la imagen del *colector* en sí.

El resto de containers se crean a partir de imágenes de terceros disponibles en internet.

4.8.2.b. Registries

Las imágenes docker se almacenan en repositorios llamados *registries*. Por conveniencia, para este proyecto se está usando el repositorio público *DockerHub*[117].

[117] https://hub.docker.com

4.8.2.c. Templates

Los manifiestos de *Kubernetes* son archivos muy verbosos en los que la información se repite en diferentes lugares.

Un formato más conveniente para escribir manifiestos son los *Helm Charts*[118]. Un *chart* es una conjunto de archivos que una vez procesados generan manifiestos de Kubernetes. Típicamente contienen un archivo *values.yml* que contiene los valores de entrada y un conjunto de manifiestos con fragmentos a sustituir por *helm*[119], el comando que realiza el "compilado".

4.8.3. Despliegue

El despliegue es la fase del pipeline en la que se aplican los manifiestos de Kubernetes con las nuevas imágenes construidas, así como la configuración.

Se realiza mediante *helm*. Esta utilidad compila los *charts* para generar manifiestos nativos de Kubernetes y hace lo que se llama una *release*. Esto no es más que registrar que ha aplicado esos manifiestos. Este registro queda almacenado en la parte servidor de *helm, tiller*, que está instalada en el clúster. Cabe mencionar que *helm* depende de *kubectl*, la API de línea de comandos de Kubernetes.

Para la autenticación entre *helm* (*kubectl* realmente) y la API de Kubernetes en el clúster se configura una *serviceAccount*. Esto no es más que unas credenciales que se crean específicamente para el CI.

4.8.4. Entorno de desarrollo

Ejecutar un clúster de *Kubernetes* en equipo portátil es posible (con herramientas como *minikube*[120]) pero no es lo más conveniente para desarrollar.

Este proyecto provee un entorno de desarrollo basado en *docker-compose*[121]. *Docker-compose* permite ejecutar los diferentes containers con la configuración adecuada sin usar *Kubernetes*. El

[118] https://github.com/kubernetes/charts
[119] https://github.com/kubernetes/helm
[120] https://github.com/kubernetes/minikube
[121] https://docs.docker.com/compose

entorno de desarrollo monta los ficheros locales como volúmen dentro del container, lo cual es muy conveniente para desarrollar y hacer cambios sin tener que reconstruir las imágenes.

En An.6 Entorno de desarrollo con docker-compose se incluyen los archivos necesarios para poner en marcha este entorno de desarrollo usando un simple comando:

```
$ docker-compose up
```

Esto iniciará los containers de todos los servicios en la máquina local. Estos servicios estarán disponibles a través de *localhost*.

4.9. Cliente

Con el objetivo de demostrar el funcionamiento del proyecto, se ha desarrollado también una aplicación cliente que realiza el envío masivo de datos al colector.

Esta aplicación está implementada en *node.js*, al igual que el colector y su código se encuentra en An.9 Cliente.

Intencionadamente, se incluye este cliente en la imagen *docker* del colector, para que se pueda ejecutar desde cualquier máquina con *docker* de una forma muy simple:

```
$ docker run -ti josefuentes/iot-collector client --help
  Usage: client [options] [command]

  Options:

    -V, --version                                output the version number
    -H, --host <name>                            set host of the service
    -P, --port <number>                          set port of the service
    -o, --origin <name>                             mark the origin of the
data
    -s, --sender <name>                             mark the sender of the
data
    -N, --namespace <name>                       mark the namespace of the
data
    -S, --silent                                 do not print traces
    -h, --help                                   output usage information

  Commands:
```

```
    healthcheck
    send <type> <value>
    incremental [options] <type> <initialValue>
    city <name>
```

Como se puede apreciar en la ayuda, el cliente tiene cuatro comandos principales.

Healthcheck contacta con el *endpoint* `/health` de la *API* y devuelve el resultado.

Send permite enviar métricas arbitrarias indicando su tipo y su valor.

Incremental es algo más complejo. Envía una serie de métricas en base a unas opciones configurables. Es obligatorio indicar el tipo y el valor inicial. Opcionalmente se puede indicar el número de eventos a enviar, tiempo que esperar entre eventos, márgenes mínimo y máximo para el valor, y un factor por el que multiplicar el valor en cada iteración.

```
$ docker run -ti josefuentes/iot-collector client incremental --help
  Usage: incremental [options] <type> <initialValue>

  Options:

     -n, --numberOfEvents <number>   number of events to send. Otherwise
will repeat forever. (default: Infinity)
     -p, --pace <seconds>              number of seconds between events
(default: 0)
      -f, --factor <factor>              positive multiplicative factor
(default: 1)
    --max <value>                   maximum value allowed
    --min <value>                   minimum value allowed
    -h, --help                      output usage information
```

City es un modo de operación que permite enviar al colector datos actuales sobre el tiempo meteorológico en una ciudad. Para eso, el cliente coge esos datos de [openweathermap.org](). Para poder usar esta funcionalidad hay que registrar una cuenta en dicho servicio y obtener un *API token*. Ese *API token* se usaría de la siguiente forma:

```
$ docker run -e "OPENWEATHERMAP_APIKEY=<API token>" -ti
josefuentes/iot-collector client city <name>
```

5. Uso y validación de la solución

5.1 Funcionamiento básico del *pipeline*

Se parte de un clúster funcional (ver Creación de un clúster de Kubernetes) sobre el que se ha instalado el *pipeline* (ver Sistema de CI/CD en el propio clúster). Se cuenta por tanto con *Gitlab* alojando el repositorio del proyecto.

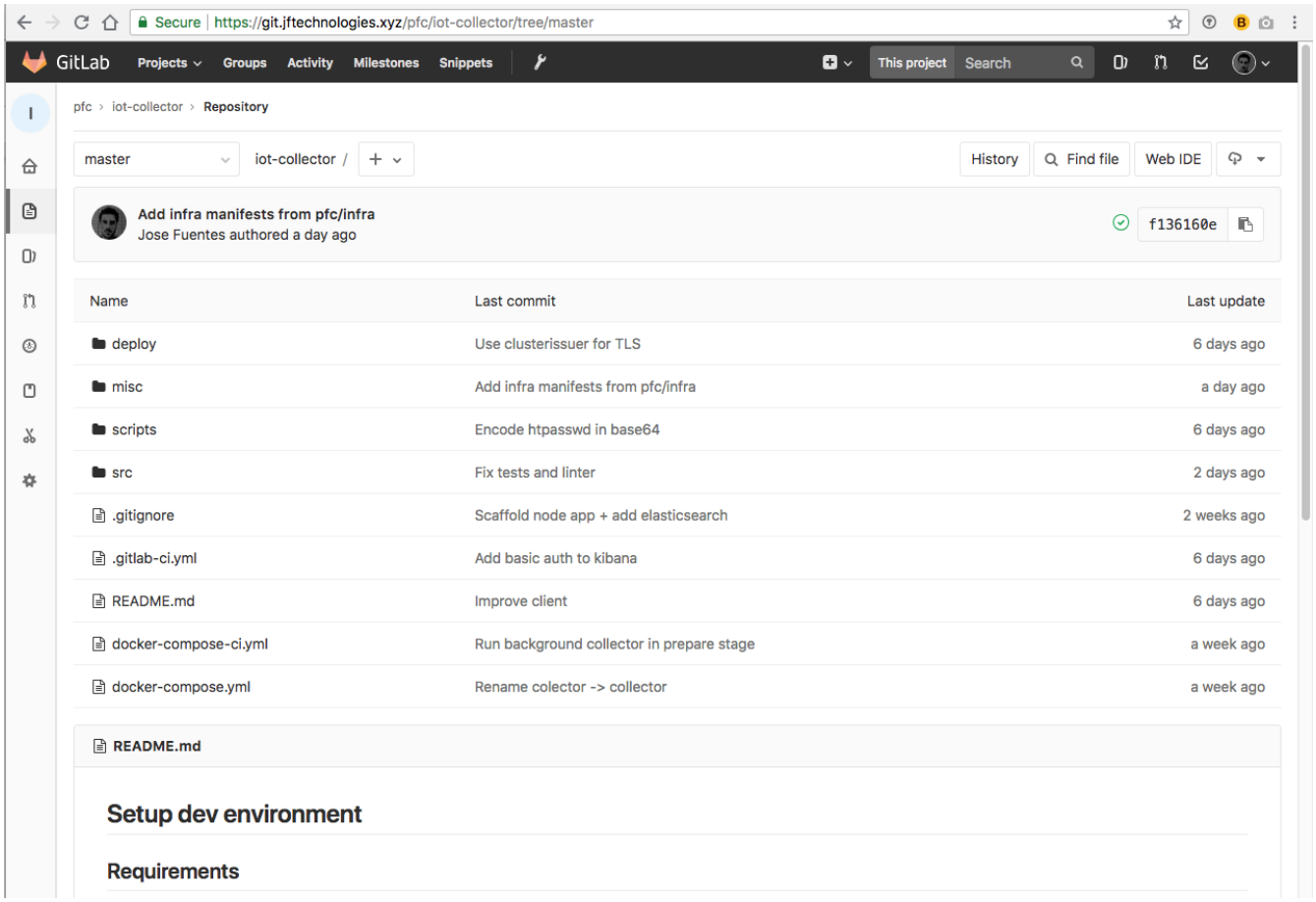


Fig 24. Vista del repositorio en *Gitlab*.

Cada cambio que se hace en el repositorio, provoca que se ejecute el pipeline con esa versión de los archivos. De forma que se obtiene un histórico con el resultado de las diferentes etapas del *pipeline* en cada cambio.

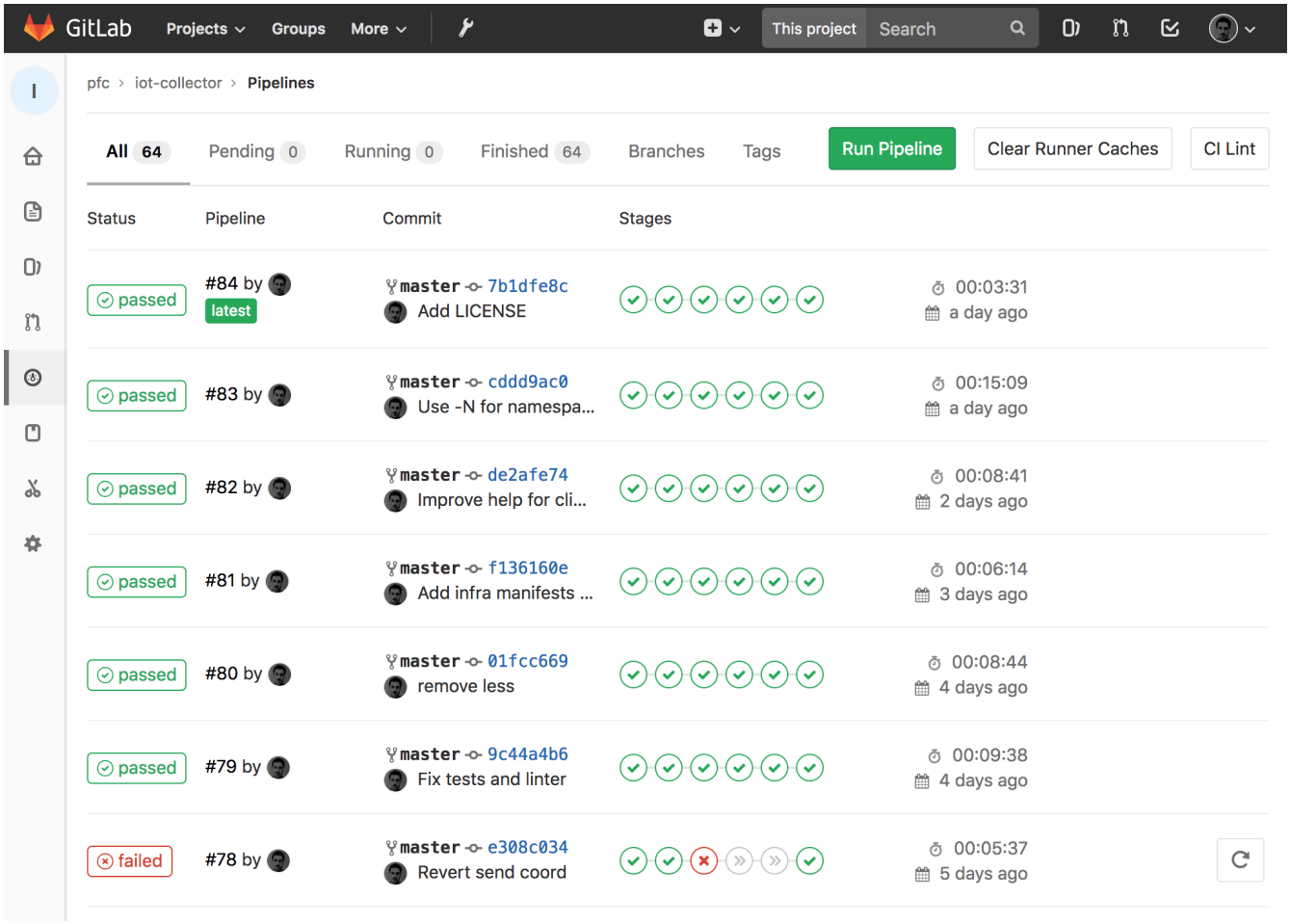


Fig 25. Vista de histórico de ejecuciones *pipeline*.

Para un cambio concreto, se pueden ver los detalles de la ejecución del *pipeline*:

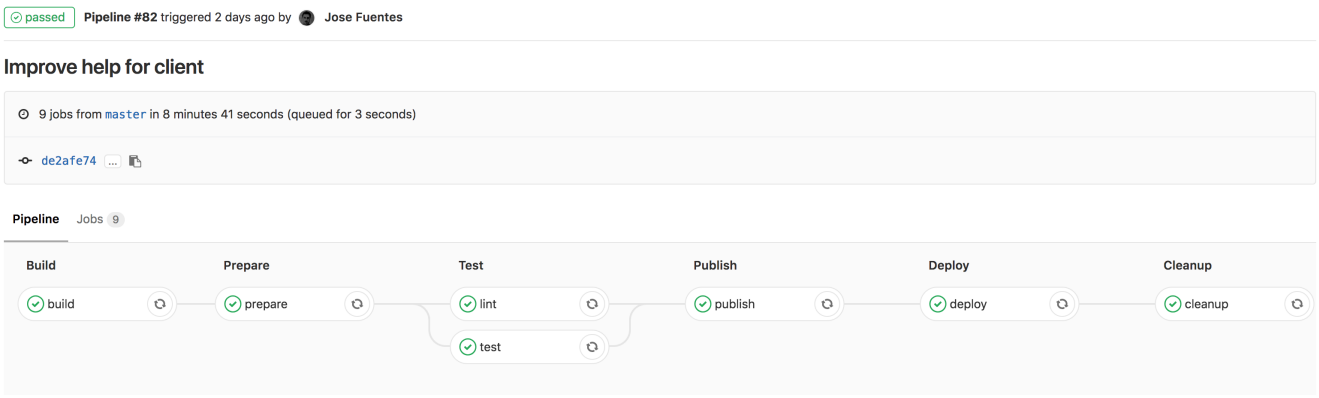


Fig 26. Detalle etapas de una ejecución del *pipeline*.

Cada uno de los elementos se denomina *job*. Para cada *job*, *gitlab runner* lanza un *pod* en *Kubernetes* donde se ejecutará el script pertinente. En la siguiente imagen se muestra el panel de control de *Kubernetes* cuando se crea uno de estos *pods*.

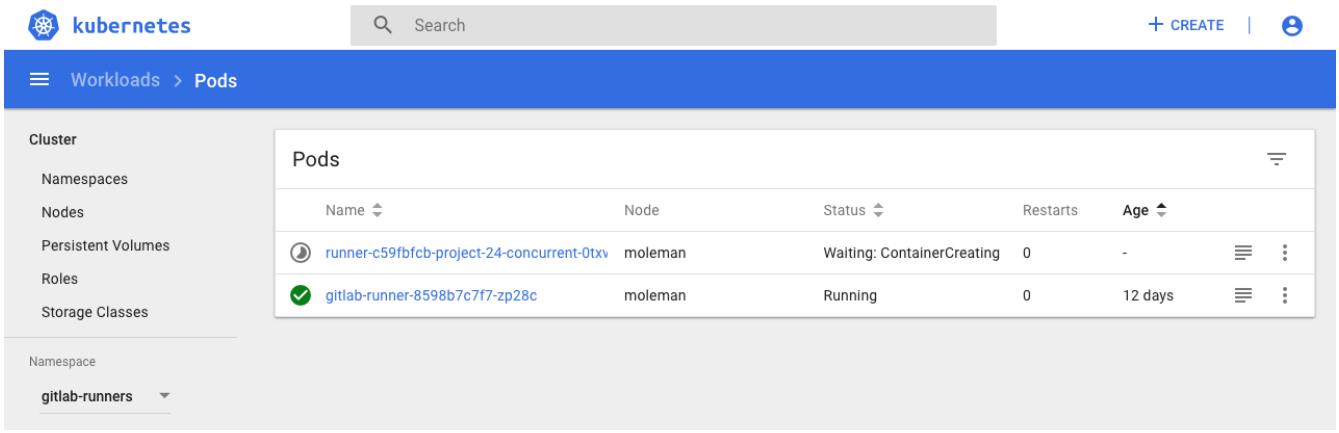


Fig 27. Nuevo *pod* para ejecutar *job* del *pipeline*.

En la interfaz de *Gitlab* se puede ver el *log* de cada uno de los *jobs*. He aquí el *log* del *job* que llamado "test", en el que se puede leer los diferentes tests realizados y el resultado de los mismos:

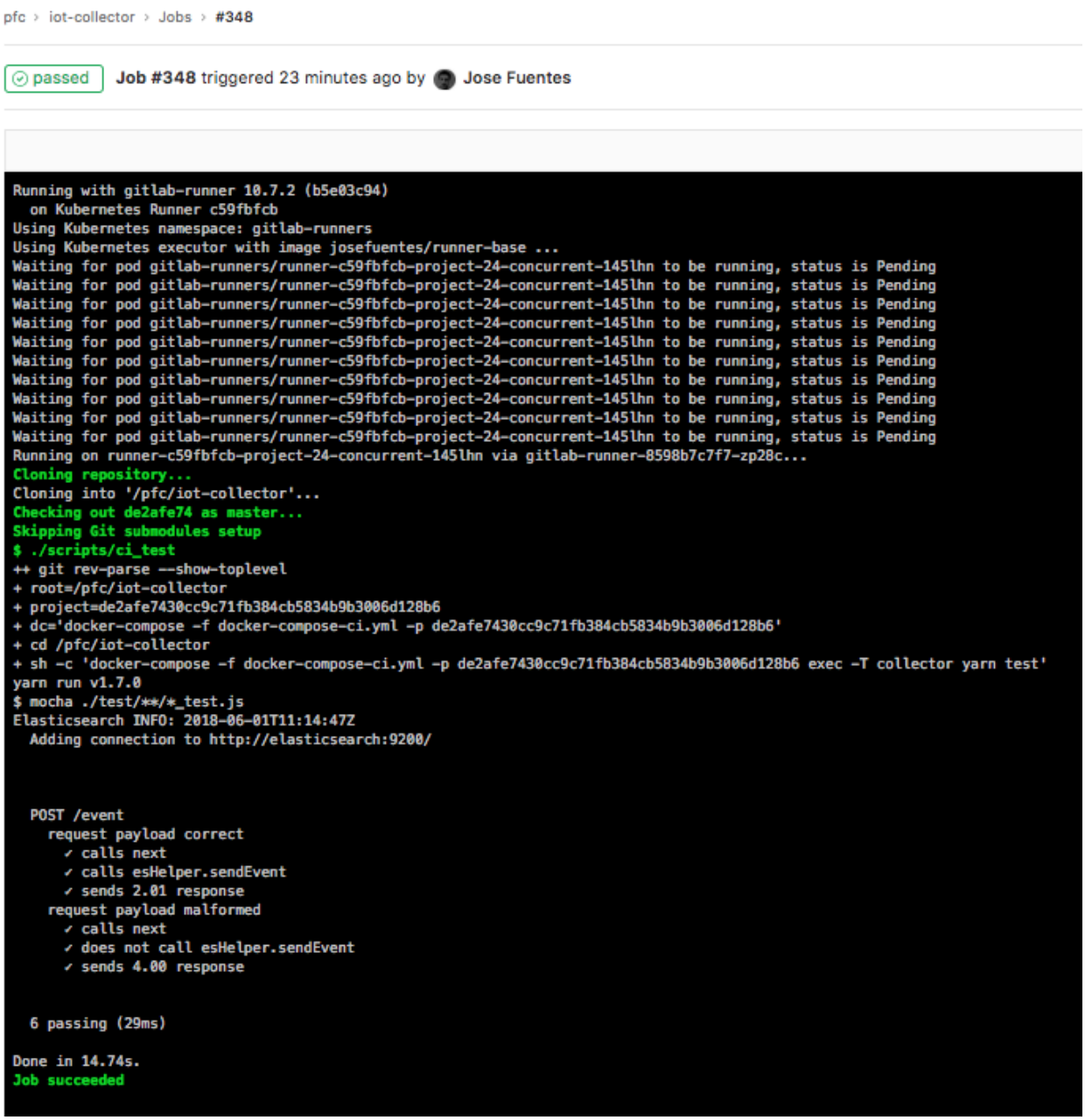


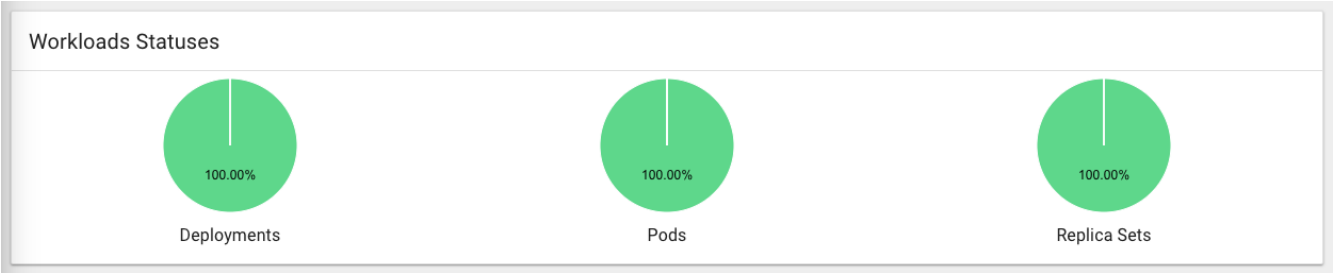
Fig 28. Log de un *job* de la etapa de "test" del *pipeline*.

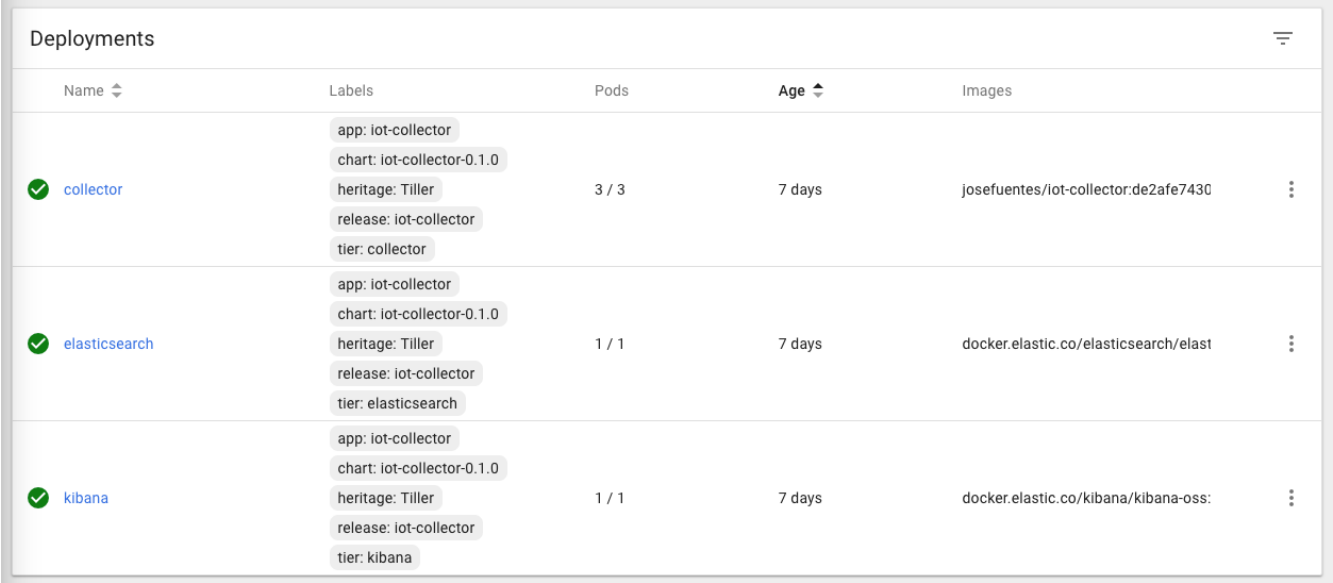
Uno de los últimos *jobs* es "deploy". Este se encarga de llamar a *helm* para desplegar la última versión de la solución en el clúster.

En caso de que ya haya una versión desplegada, *Kubernetes* analiza los cambios requeridos. Cuando cambia la imagen, por defecto *Kubernetes* realiza un *rolling upgrade*. Esta estrategia consisten en ir sustituyendo las réplicas de los *pods* una por una por la versión nueva, comprobando que tras cada paso todo funciona sin errores. De esta forma, cada vez que se hace un cambio en el

colector, por ejemplo, se aplicará ese cambio gradualmente sobre las réplicas. Durante el proceso hay realmente dos versiones de la aplicación ejecutándose. En este caso, el balanceador de carga suele estar configurado para aplicar afinidad según origen. Esto significa que intentará llevar al mismo *pod* a las peticiones que vengan del mismo origen. Puesto que se todo esto ocurre detrás del *firewall* del router que da acceso a internet (recordar que se hace *NATP*), la única forma de diferenciar el origen sería en la capa 7. Esto los balanceadores de carga *HTTP* lo manejan perfectamente. Como el colector usa *CoAP*, esta afinidad no va a funcionar.

En la siguiente imagen se muestran todos los recursos del *namespace* "iot-collector", que es donde se despliega la solución. Como se puede apreciar, existen tres réplicas del *pod* colector y el servicio del mismo tiene asignada una de las IPs gestionadas por *MetalLB* (ver [Soporte para Kubernetes LoadBalancers]):





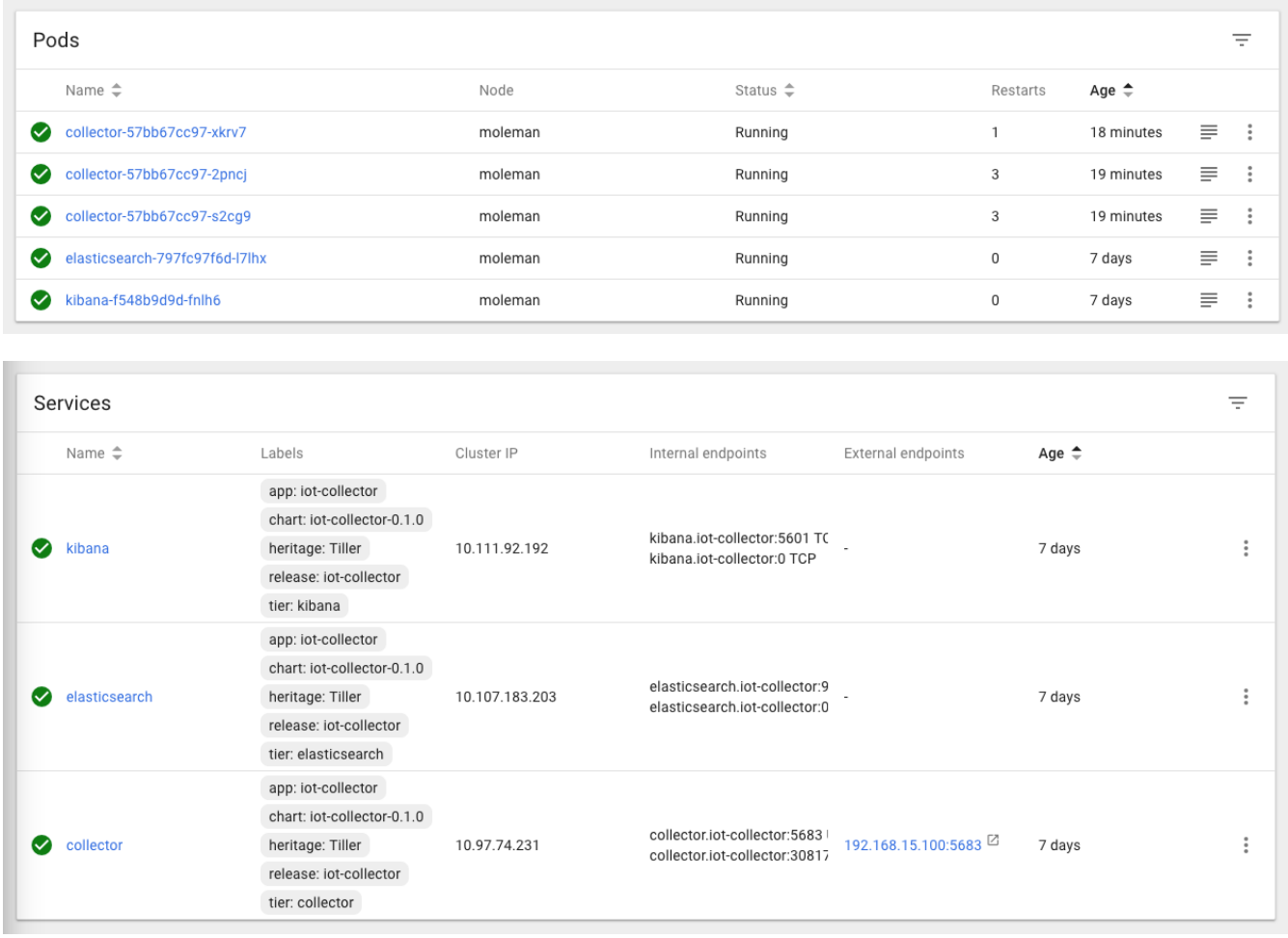


Fig 29. Recursos de la solución mostrados por el *dashboard* de *Kubernetes*.

5.2. Envío de eventos con el cliente y búsqueda de datos en *Kibana*

Se pretende demostrar el funcionamiento básico del servicio usando el cliente implementado (ver Cliente). El siguiente comando envía diez eventos de tipo "temperature" con empezando con valor 15 e incrementando un 10% en cada una. El servidor está accesible en *collector.moleman.jftechnologies.xyz*:

```
$ docker run -ti josefuentes/iot-collector client -H
collector.moleman.jftechnologies.xyz -s demoMemoria -o probe1 incremental
--numberOfEvents 10 -f 1.1 temperature 15
-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
```

```
  payload: { type: 'temperature', value: 15 },
  timestamp: '2018-06-01T14:08:24.771Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
  payload: { type: 'temperature', value: 16.5 },
  timestamp: '2018-06-01T14:08:24.772Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
  payload: { type: 'temperature', value: 18.150000000000002 },
  timestamp: '2018-06-01T14:08:24.774Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
  payload: { type: 'temperature', value: 19.965000000000003 },
  timestamp: '2018-06-01T14:08:24.816Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
  payload: { type: 'temperature', value: 21.961500000000004 },
  timestamp: '2018-06-01T14:08:24.817Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
  payload: { type: 'temperature', value: 24.157650000000007 },
  timestamp: '2018-06-01T14:08:24.818Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
```

```
  origin: 'probe1',
  payload: { type: 'temperature', value: 26.57341500000001 },
  timestamp: '2018-06-01T14:08:24.819Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
  payload: { type: 'temperature', value: 29.230756500000016 },
  timestamp: '2018-06-01T14:08:24.819Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
  payload: { type: 'temperature', value: 35.369215365000024 },
  timestamp: '2018-06-01T14:08:24.821Z' }
----------

-- POST: /event -> 2.01 --
 { namespace: 'default',
  sender: 'demoMemoria',
  origin: 'probe1',
  payload: { type: 'temperature', value: 32.15383215000002 },
  timestamp: '2018-06-01T14:08:24.820Z' }
----------
```

Se puede apreciar que se imprimen en el log desordenados. Esto es porque los eventos se envían asíncronamente, es decir, antes de que se termine de enviar uno, ya se está enviando el siguiente. Se va imprimiendo en el *log* conforme las respuestas del servidor se van recibiendo.

Para consultar las medidas, se puede acceder a la sección "Discover" de *Kibana*, que permite hacer búsquedas en base a los parámetros indexados. Cómo se ha establecido "`probe1`" como origen, se pueden filtrar los eventos anteriores con la consulta "`origin:probe1`". *Kibana* muestra un gráfico temporal representando los momentos en los que ocurrieron los eventos y una lista con todos ellos. Se puede limitar la búsqueda a un rango temporal.

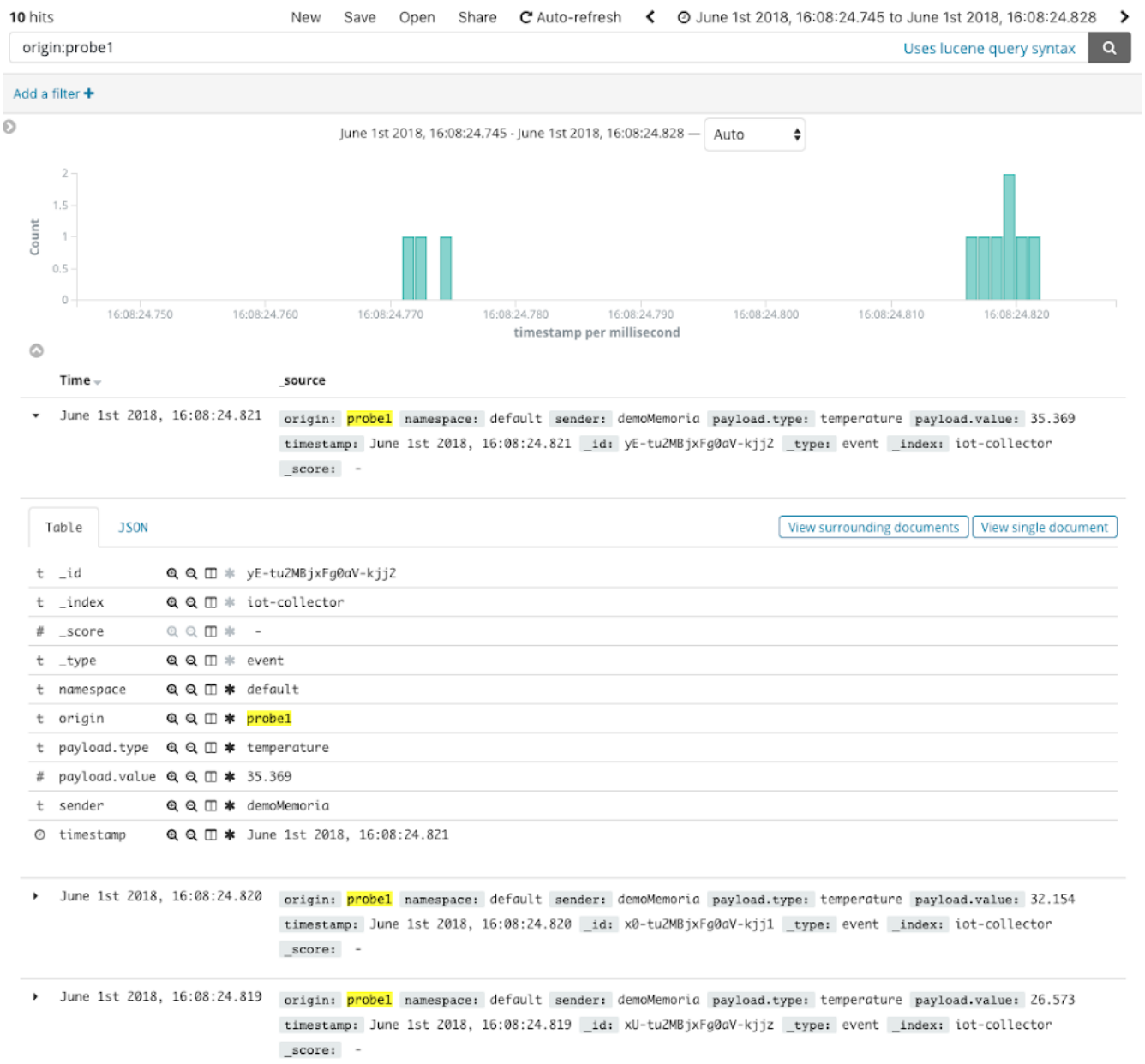


Fig 30. Visualización de eventos "en crudo" usando *Kibana*.

5.3. Ejemplos de *dashboards* con datos recolectados periódicamente

Para hacer un demostración algo más ambiciosa, se han creado en *Kubernetes* varios *CronJobs* que envían medidas al colector de forma periódica. Un *CronJob* es un recurso de *Kubernetes* que permite hacer que determinadas operaciones se ejecuten periódicamente siguiente el formato *cron*[122]. Estos

[122] http://www.nncron.ru/help/EN/working/cron-format.htm

CronJobs se definen en An.10 CronJobs para el envío periódico de medidas y envían datos meteorológicos de varias ciudades"

Los datos enviados se pueden identificar porque tienen como `"sender"` `"weather-city"` y como `"origin"` el nombre de la ciudad.

Kibana tiene un módulo especializado en representar series temporales de eventos. Usa un lenguaje propio para describir dichos gráficos. Este módulo se llama *Timelion*[123]. La figura 31 muestra como se puede usar *Timelion* para comparar la temperatura de varias ciudades durante los últimos cuatro días.

Kibana permite elaborar *dashboards* conjugando diferentes tipos de *widgets*. Uno se esos tipos son los gráficos provenientes de *Timelion*. En la figura 32 se ha realizado un panel que muestra los datos meteorológicos filtrados por ciudad.

[123] https://www.elastic.co/guide/en/kibana/current/timelion.html

Fig 31. Gráfica a partir de series temporales con *Timelion*.

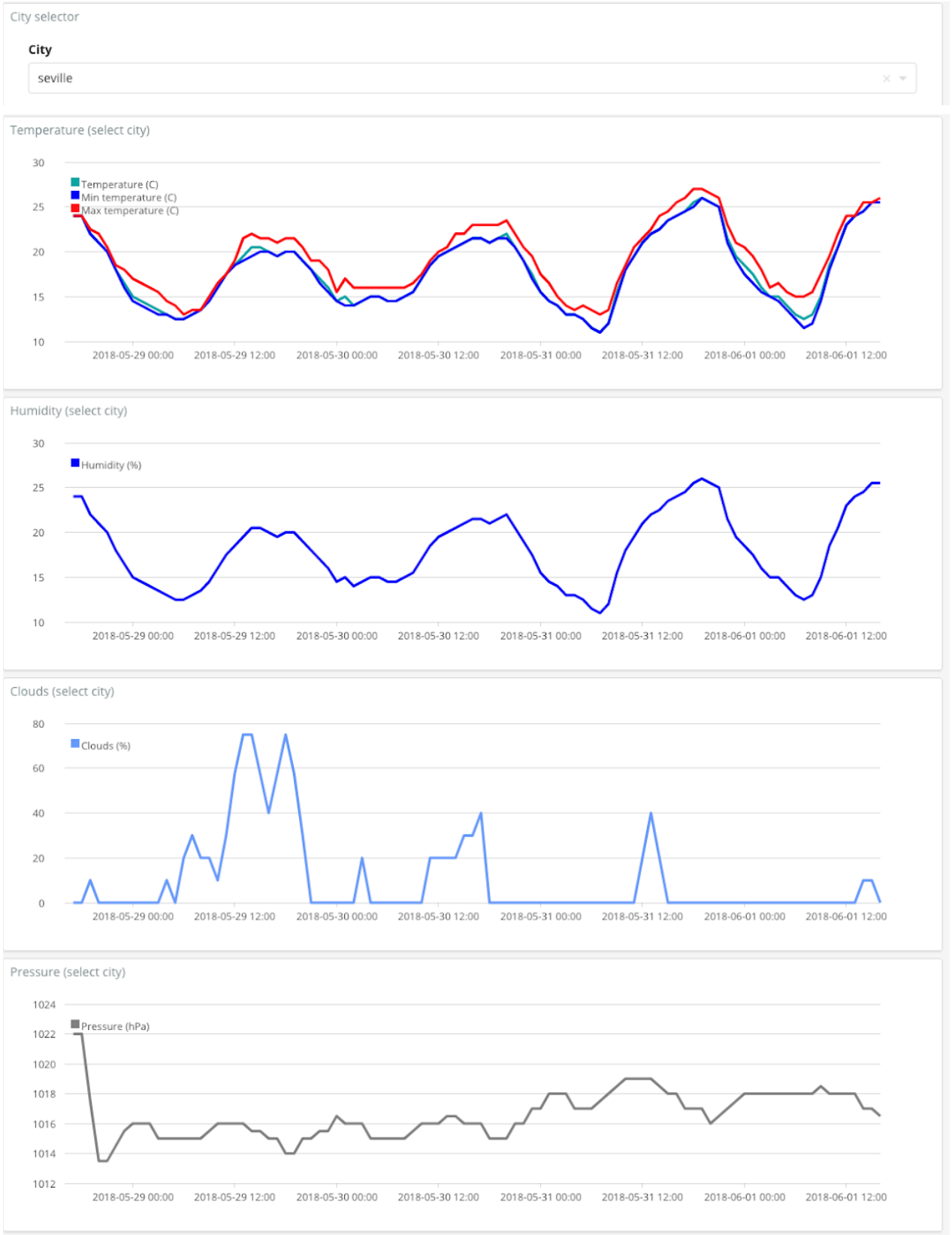


Fig 32. *Dashboard* de datos meteorológicos de una ciudad.

5.4. Pruebas de rendimiento

Con el fin de validar que la solución escala, se ha instrumentado el código del colector para que envíe información a *Elasticsearch* tras servir cada petición. De esta forma se tienen métricas del código de estado de la respuesta o de cuánto tiempo se ha tardado en procesar.

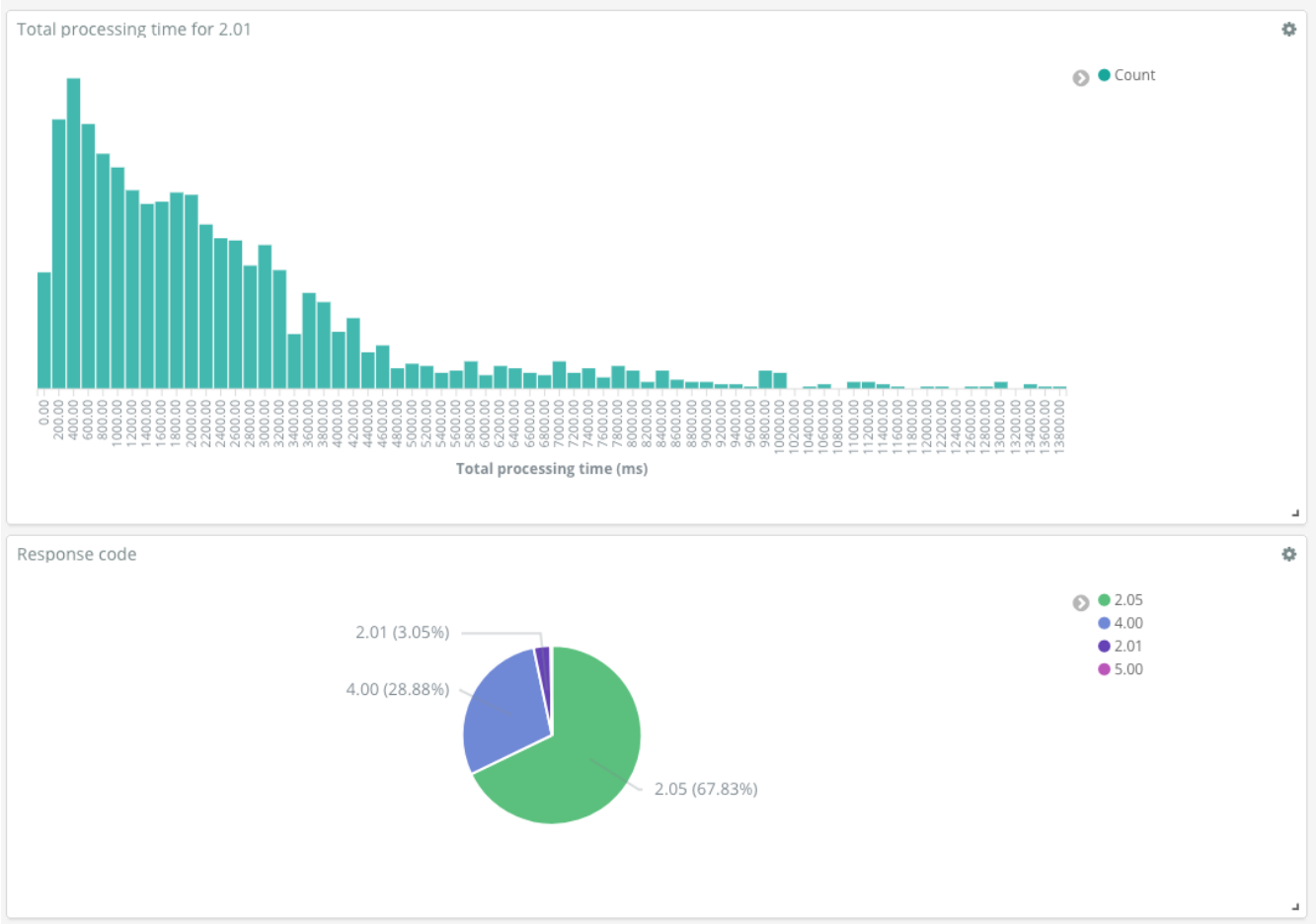


Fig 33. Ejemplo de métricas de las peticiones al colector.

Se ha realizado una prueba consistente en enviar continuamente peticiones concurrentes a la API desde un cliente externo durante quince minutos. Primero se ha hecho con una sola réplica del colector, y posteriormente con tres. Usando *Kibana* se han calculado los percentiles del tiempo de procesado de las peticiones servidas con éxito.

Para la prueba con una sola réplica estos son los resultados:

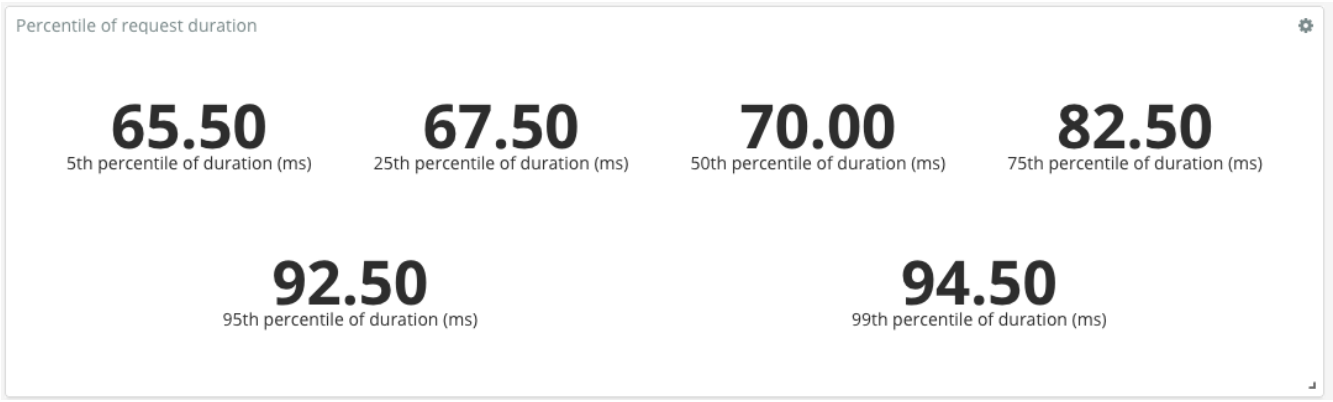


Fig 34. Percentiles del tiempo de respuesta con un sólo nodo.

Para la prueba con tres réplicas estos son los resultados:

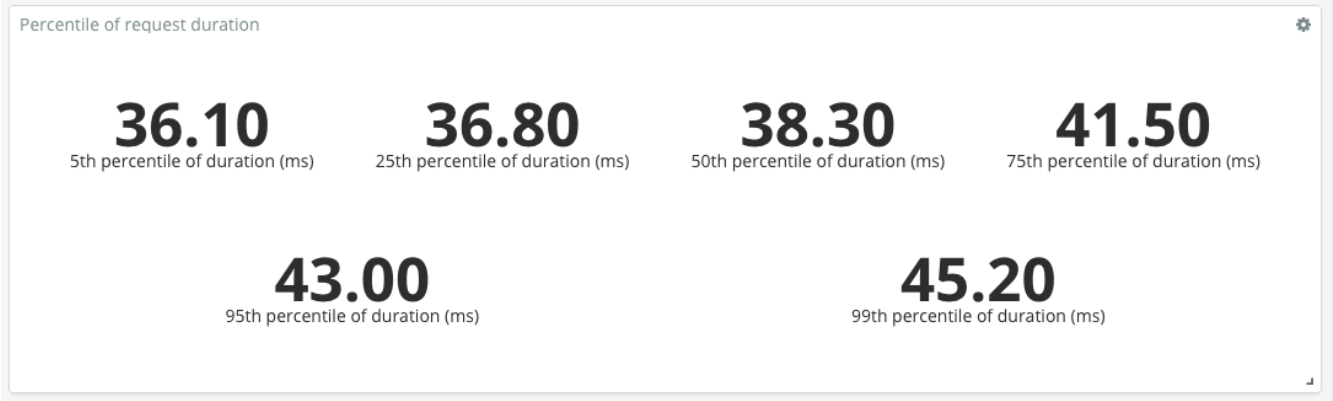


Fig 35. Percentiles del tiempo de respuesta con tres nodos.

Se aprecia como se ha reducido sensiblemente el tiempo de procesamiento al aumentar el número de réplicas.

5.5. Validación de especificaciones

En las siguientes secciones se valida el cumplimiento de los requisitos enumerados en el apartado 3.6. Tabla resumen de especificaciones.

5.5.1. Aplicación Cloud Native

Para medir el tiempo de carga de la aplicación, basta con acceder a los *log* de uno de los *pods* que sirve el colector. *Kubernetes* guarda los *logs* de todos los pods, y añade una huella temporal a cada

línea. En la figura 36 se muestra el log en cuestión. Se puede ver que entre la primera línea (inicio del proceso) y la línea que contiene "`CoAP server listening on 0.0.0.0:5683`" han pasado sólo 2,12 segundos, lo que valida la condición de que la aplicación arranque en menos de 3 segundos (requisito 1.1). El proceso *node.js* es *mono thread*[124], por lo que nunca va a usar más de una CPU.

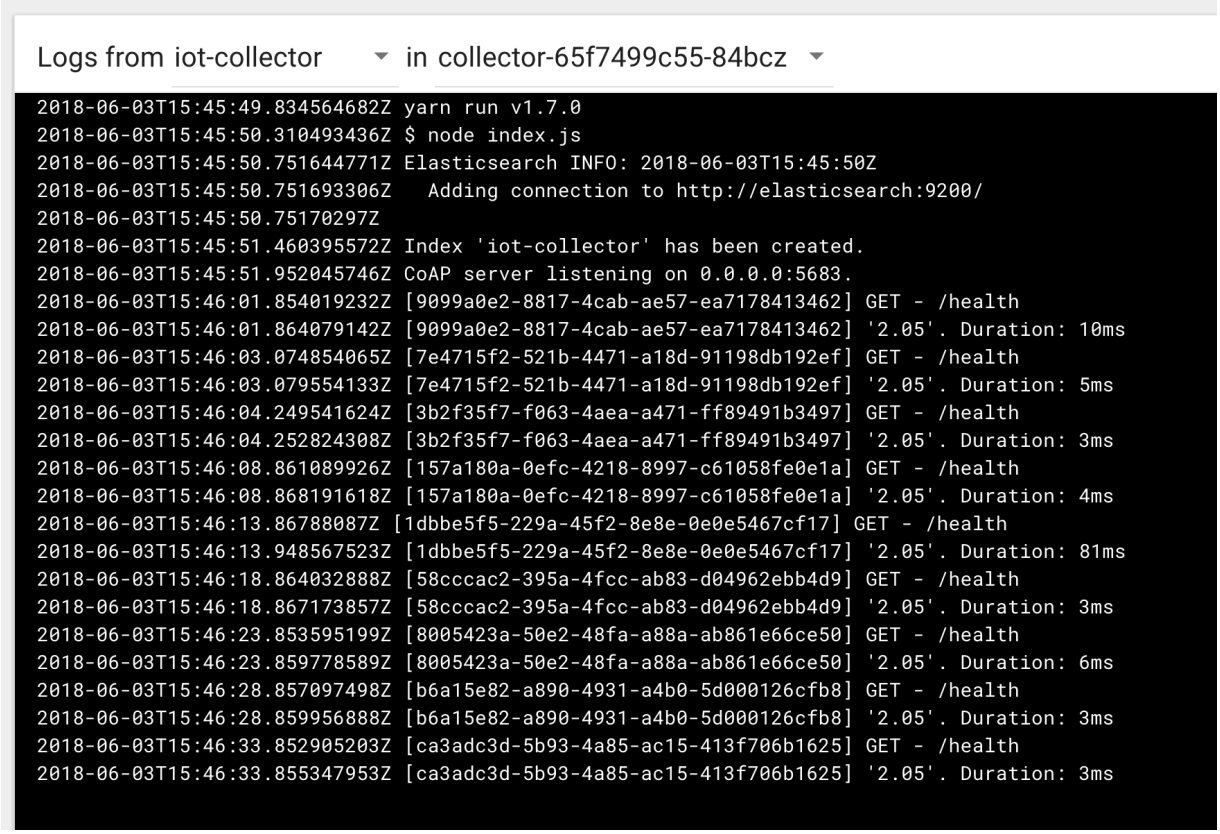


Fig 36. Log de inicio de colector.

Por defecto *node.js* atiende a la señal *SIGTERM*[125]. Se esperará a que se cierren los recursos abiertos. Esto satisface el requisito 1.2.

Durante este capítulo se ha demostrado que se satisface el requisito 1.3, soporte de paralelismo, puesto que la aplicación ha funcionado correctamente con tres réplicas del *pod* colector.

En la sección 4.6.1 se muestra como la etapa de base de datos tiene un volúmen en el que se guarda el estado de la misma. Para comprobar que la persistencia funciona (requisito 1.4), se ha probado a hacer un reinicio de todos los nodos del clúster, y se ha comprobado que no se han perdido datos.

[124] Dícese de un proceso que sólo puede usar un hilo de la CPU.
[125] https://nodejs.org/api/process.html

El requisito 1.5 exige el despliegue de la solución en base a manifiestos. El *job* del *pipeline* llamado "deploy" demuestra que la solución se despliega usando el *helm chart* (An.8 Helm chart) con un sólo comando.

Inspeccionando el código fuente (An.4 Código fuente del colector) se puede verificar que en ningún punto se accede directamente a ningún dispositivo hardware (requisito 1.6). De hecho, todos los directorios a los que se accede están dentro del sistema de ficheros del container.

5.5.2. Configuración de la infraestructura

El requisito 2.1 queda cubierto con la configuración del plano de control de *Kubernetes* descrita en la sección 4.3.2. Sistema operativo y configuración básica de las máquinas.

La comunicación entre contenedores en diferentes nodos (requisito 2.2) queda resuelta en la sección 4.3.4. Instalación de la capa de red de Kubernetes.

El requisito 2.3 trata sobre la resolución interna de nombres por DNS. Junto con el plano de control de *Kubernetes* (4.3.3. Instalación del plano de control de Kubernetes) se despliega por defecto un servidor DNS. La capa de red recientemente mencionada se integra con dicho servidor DNS para proveer las direcciones de los servicios y *pods* dentro de la red virtual.

El requisito 2.4 implica que las comunicaciones dentro del cluster sean seguras. La capa de red ya mencionada configura *netfilter*[126] para que las comunicaciones sólo sean posibles entre los recursos definidos en *Kubernetes*.

La instalación y configuración del plano de control (requisito 2.5) se aborda en la sección 4.3.3. Instalación del plano de control de Kubernetes.

5.5.3. *Pipeline* de desarrollo

El requisito de integración continua (requisito 3.1) queda satisfecho con las etapas "Build", "Prepare", "Test" y "Publish" mostradas el la sección 5.1 Funcionamiento básico del pipeline. El despliegue continuo (requisito 3.2) con las etapas "Publish" y "Deploy" de la misma sección.

[126] https://www.netfilter.org/: es el firewall integrado en *Linux*.

5.5.4. Funcionalidades de la aplicación

El requisito 4.1 (presentar una API externa) queda demostrado con la interacción cliente-servidor demostrada en este capítulo.

El requisito 4.2 (protocolos adecuados para *IoT*) lo satisface el hecho de que la aplicación usa *UDP* en el nivel de transporte y *CoAP* en el nivel de aplicación.

Por diseño, la aplicación procesa los datos al vuelo, sin guardarlos en almacenamientos intermedios. Esto satisface el requisito 4.3.

Para solventar la capa de visualización (requisito 4.4) se ha incluido *Kibana*. Se ha demostrado en esta sección que es una capa de visualización muy versátil y adecuada para representar series temporales de eventos.

5.5.5. *Open-source*

Se puede comprobar que todas las tecnologías y dependencias usadas en este proyecto tienen licencias consideradas *open-source* (requisito 5.1).

6. Conclusiones

Durante el desarrollo de este trabajo se ha observado que existe una clara diferencia entre ejecutar un servicio en el *cloud* y diseñar el servicio a conciencia para que sea *cloud native*. La primera estrategia es posible, y es la estrategia a seguir si no se desea rediseñar por completo una solución. La segunda estrategia contempla tener en cuenta gran cantidad de restricciones que implican a los niveles más bajos del diseño, por lo tanto es más fácil de aplicar cuanto antes de ataque en el proceso.

En cuanto a los beneficios de ser *cloud native*, se facilita enormemente la operación del sistema: no hay máquinas que mantener, actualizar o parchear; hay muchas menos fricciones entre el entorno de desarrollo y producción; y se llega a sistemas mucho más fáciles de gestionar, operar y escalar.

En cuanto a *Kubernetes*, parece ser una plataforma más o menos madura, que puede ser una base sólida sobre la que construir soluciones de casi cualquier tipo. Se aprecia que casi la totalidad de los

cloud providers están ofreciendo opciones para ejecutar *Kubernetes* en sus plataformas, lo cual enriquece el ecosistema porque las soluciones son mucho más fáciles de portar de un sitio a otro, lo que a su vez mejora la competitividad.

En cuanto al uso de *containers* como *backend* para servicios *IoT*, se ha demostrado que ofrecen ventajas interesante sobre todo en la etapa más externa, la de recolección de datos. La tecnología de *cloud* basada en *containers* permite ajustar rápidamente el tamaño de esta etapa, haciendo muy flexible la capacidad de ingestión de datos. Los sistemas multietapa (ingestión, procesado, almacenamiento) como el aquí diseñado parecen ser un caso de uso común para el mundo del *IoT*, y los *containers* son el elemento constructivo básico para ese tipo de soluciones.

Por último se ha observado que, aunque el desarrollo y adopción de *containers* en el *cloud* (específicamente de *Kubernetes*) está creciendo a un ritmo muy acelerado, hay algunos aspectos que indican aún falta de madurez. Por ejemplo, el soporte para servicios que se comunican sobre *HTTP* está muy avanzado, pero es común encontrar problemas, falta de documentación y de funcionalidades para otros protocolos, como se ha comprobado con CoAP para el que no existe soporte de *Ingress* en *Kubernetes*.

7. Trabajo futuro

Una evolución de este trabajo sería añadir monitorización a nivel de aplicación. Si esta solución se usara para un producto serio en producción, sería conveniente instrumentar todo el código para enviar métricas que permitan monitorizar el estado del servicio. Ejemplos de métricas útiles serían el número de peticiones, el tiempo que tardan en ser servidas o el código de estado que devuelven. La monitorización es vital para identificar problemas antes de que sean graves y evitar interrupciones del servicio. También son importantes para los desarrolladores ya que permiten evaluar si un cambio ha introducido algo que hace empeorar el rendimiento, o simplemente se ha introducido un fallo no capturado por los tests que hace que los usuarios reciban errores.

Como complemento a la monitorización, se sugiere añadir un sistema de alertas. Estos sistemas permiten definir umbrales y condiciones sobre las métricas. En caso de que esos umbrales se sobrepasen, el sistema se encarga de hacerlo saber a la persona al cargo.

Otra posible evolución de este proyecto en podría incluir mejoras como la automatización del escalado en cada una de las etapas. La implementación actual exige que un humano tome la decisión de crear más réplicas en la etapa de colectores si la cantidad de peticiones es demasiado grande. No sería complicado hacer una integración entre el sistema de monitorización y el plano de control de *Kubernetes* de forma que se esté continuamente monitorizando el número de peticiones y los recursos consumidos y se automáticamente se provisionen nuevas instancias si la carga aumenta o se destruyan las existentes cuando la carga disminuya.

En cuanto a la construcción de imágenes, a día de hoy es un problema ejecutar *docker* sobre *docker*. A menudo es necesario recurrir a trucos como el usado en este proyecto (ejecutar en segundo demonio *docker* en la misma máquina). Estas configuraciones pueden llegar a ser problemáticas y a menudo son desaconsejadas incluso por sus creadores[127]. Últimamente hay una tendencia por usar *Bazel*[128] para construir imágenes *docker*[129]. *Bazel* es un potente *build system* creado por *Google* y usado por la mayoría de sus grandes proyectos *open-source* como el propio *Kubernetes* o *TensorFlow*. *Bazel* proporciona *bindings*[130] de las operaciones que hace el demonio *docker* al interpretar un *Dockerfile* y permite replicar el proceso en un entorno aislado y controlado. Sería interesante aplicar este concepto al *pipeline* desarrollado en este proyecto de forma que se pudiera reemplazar el demonio *docker* secundario dedicado a ejecutar construir imágenes con *bazel* ejecutándose sobre un container normal, sin privilegios y totalmente aislado.

Por último, se plantea mejorar el soporte de *Kubernetes* para el protocolo *CoAP*. Sería interesante desarrollar soporte para *Ingresses CoAP*. Esto se podría implementar sobre la implementación con *nginx* para *IngressController*.

8. Glosario

API: *Application Programming Interface*. Término usado para referirse a la interfaz definida por un software para interactuar con el.

Bare metal: término usado para referirse a sistemas que se ejecutan sobre hardware real, no virtual.

[127] http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci
[128] https://bazel.build
[129] https://medium.com/bitnami-perspectives/building-docker-images-without-docker-c619061b13a9
[130] En este contexto un *binding* es un adaptador que permite hacer una operación nativa de un tipo de tecnología en otro.

Container: término que designa un contexto de recursos de sistema, sistema de ficheros y espacio de procesos y usuarios limitados en cierto modo por el sistema operativo.

m2m: *Machine to Machine*. Término acuñado para describir comunicaciones que tienen como origen y como destinatario una máquina en lugar de un humano.

UNIX-like: término usado para referirse a sistemas operativos similares al sistema UNIX pero que no necesariamente cumplen con la *Single UNIX Specification*[131].

REST: *REpresentational State Transfer*. Se dice que un servicio es *REST-compliant* cuando cumple ciertas restricciones como que tenga una arquitectura cliente-servidor, que el cliente no tenga que mantener contexto entre peticiones, o que el contenido sea cacheable.

Runtime: puede referirse simplemente al momento en el que la aplicación se ejecuta. Por extensión, también se usa para referenciar al contexto formado por todas las librerías, dependencias de sistema e intérprete (en caso de haber alguno) que una aplicación requiere para funcionar.

Serverless Paradigma de computación en el que se abstrae al usuario por completo del hardware, y sistema operativo, de forma que ofrece un contexto en el que son ejecutadas las funciones que el usuario define. También se denomina *FaaS* (*Function As A Service*).

SLA: *Service Level Agreement*. Compromiso acordado entre un proveedor de servicio y un cliente que expresa ciertas cotas de calidad, disponibilidad y responsabilidad con respecto a dicho servicio.

SND: *Software Defined Networking*. En sistemas con máquinas y redes virtuales, son redes que se definen a nivel lógico sobre el hardware.

VM: *Virtual Machine* o *Máquina Virtual* en español. Es un término usado para referirse a una abstracción lógica de recursos de computación sobre los que se puede ejecutar un sistema operativo.

WLAN: Wireless Local Access Network. Red de alcance local (del orden de decenas de metros) en la que los dispositivos se conectan de forma inalámbrica.

WPAN: *Wireless Personal Access Network*. Red de alcance personal (del orden de varios metros) en la que los dispositivos se conectan de forma inalámbrica.

[131] http://www.unix.org/what_is_unix/single_unix_specification.html

WAN: *Wide Access Network*. Red de amplio alcance.

9. Bibliografía y referencias

Conceptos generales de "*IoT*" y su origen

→ "'*Internet of things*' *topic*" de *Google Trends* (https://trends.google.com/trends/explore?date=2010-02-01%202018-01-20&q=%2Fm%2F02vnd10). Consultado en Enero de 2018.
→ "The Origin of the Internet of Things" por *Duncan McFarlane* (https://www.redbite.com/the-origin-of-the-internet-of-things/). Consultado en Enero de 2018.
→ "How the world's first webcam made a coffee pot famous" por *Rebecca Kesby, BBC*. (http://www.bbc.com/news/technology-20439301). Consultado en Enero de 2018.
→ Auto-ID labs homepage, (https://autoidlabs.org). Consultado en Enero de 2018.

Conceptos generales de "*Cloud*" y su origen

→ "*EC2 origins*" por *Benjamin Black* (http://blog.b3k.us/2009/01/25/ec2-origins.html). Consultado en Noviembre de 2017.
→ "Quarterly revenue of AWS" (https://www.statista.com/statistics/250520/forecast-of-amazon-web-services-revenue/). Consultado en Noviembre de 2017.
→ "About AWS" (https://aws.amazon.com/about-aws/). Consultado en Noviembre de 2017
→ "What is cloud computing" (https://aws.amazon.com/what-is-cloud-computing/). Consultado en Noviembre de 2017.
→ "*Containers at Google*" (https://cloud.google.com/containers/). Consultado Noviembre 2017.

Containers

→ "*Understanding the docker internals*" por *Nitin Agarwal* (https://medium.com/@nagarwal/understanding-the-docker-internals-7ccb052ce9fe). Consultado en Noviembre de 2017.
→ "*A Gentle Introduction To Docker And All Things Containers*" por *Jérôme Petazzoni* (https://www.slideshare.net/jpetazzo/introduction-docker-linux-containers-lxc). Consultado en Noviembre de 2017.

→ "*Realizing Linux Containers (LXC)*" por *Boden Russell* (https://www.slideshare.net/BodenRussell/realizing-linux-containerslxc). Consultado en Noviembre de 2017.
→ "*Introduction to LXC*" (https://linuxcontainers.org/lxc/introduction/). Consultado en Noviembre de 2017.
→ "*Control Groups (cgroups) for the Web?*" por *Ilya Grigorik* (https://www.igvita.com/2016/03/01/control-groups-cgroups-for-the-web/). Consultado en Noviembre de 2017.
→ "*Resource limiting using cgroups*" (http://hintcafe.net/post/60223405371/resource-limiting-using-cgroups). Consultado en Noviembre de 2017.
→ "*Docker security*" (https://docs.docker.com/engine/security/security/). Consultado en Noviembre de 2017.
→ "*Introduction to LXC*" (https://linuxcontainers.org/lxc/introduction/). Consultado en Noviembre de 2017.

Elección de la base de datos

→ "*TSDB Explained*" por *Influxdata* (https://www.influxdata.com/time-series-database/). Consultado en Marzo de 2018.
→ "*What the heck is time-series data (and why do I need a time-series database)?*" por *Ajay Kulkarni* (https://blog.timescale.com/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563). Consultado en Marzo 2018.
→ "*Top 10 time series databases*" por *Steven Acreman* (https://blog.outlyer.com/top10-open-source-time-series-database). Consultado en Marzo de 2018.

Recursos generales de *Kubernetes*

→ "*Kubernetes Componentes*" (https://kubernetes.io/docs/concepts/overview/components). Consultado en Enero de 2018.

→ "*Container Networking Interface*" (https://github.com/containernetworking/cni/blob/master/SPEC.md). Consultado en Enero de 2018.
→ "*Kubernetes Reference*" (https://kubernetes.io/docs/reference). Consultado en Enero de 2018.

CoAP

→ "*RFC 7252: The Constrained Application Protocol (CoAP)*" (https://tools.ietf.org/html/rfc7252). Consultado en Enero de 2018.

Otros

→ "*Cloud Native Infrastructure*", por *Justin Garrison* y *Kris Nova*. Editorial *O'Reilly*, 2018.
→ "A Runtime System", por *Andrew W. Appel*. *Princeton University, 1989*.

10. Índice de figuras

11. Índice de anexos

Se adjunta a este documento el repositorio *git* con el código fuente del proyecto. Este contenido se disecciona en anexos en esta sección en forma de índice, con la intención de facilitar al lector la localización de los archivos.

An.1 Manifiestos Gitlab para Kubernetes

Conjunto de manifiestos que permiten desplegar y configurar *Gitlab* sobre el clúster con *kubectl*.

Se encuentran en el directorio `source/misc/k8s-gitlab`.

An.2 Dockerfile para imagen base de *Gitlab runner executors*

Dockerfile para generar la imagen base usada por los *executors* que *Gitlab runner* lanza para ejecutar los *jobs* del CI y script construye y publica la imagen.

Se encuentran en el directorio `source/misc/gitlab-runner-base`.

An.3 Script de inicio de demonio *docker* secundario

Script que inicia un segundo demonio *docker*, usado como *sandbox* para construir las imágenes *docker* en el *CI*.

Se encuentra en `source/misc/secondary_docker/start.sh`.

An.4 Código fuente del colector

Código fuente de la aplicación *node.js* que sirve la *API CoAP*.

Se encuentra en el directorio `source/src`.

An.5 Dockerfile para construir imagen del colector

Dockerfile para generar la imagen docker del colector de datos.

Se encuentra en `source/src/Dockerfile`.

An.6 Entorno de desarrollo con docker-compose

Archivo docker-compose que permite levantar el entorno de desarrollo y archivos de configuración para elasticsearch y kibana en dicho entorno.

El manifiesto de docker-compose se encuentra en `source/docker-compose.yml`.

Las configuraciones se encuentran en `source/deploy/config`.

An.7 *Pipeline*

Definición del *pipeline* CI/CD y scripts que usa.

La definición de las etapas del *pipeline* están en el archivo `source/.gitlab-ci.yml`.

Esos pasos llaman a scripts que están en `source/scripts`.

An.8 *Helm chart*

Archivos que componen el *chart*, paquete que permite instalar el solución de forma fácil sobre *Kubernetes*.

Se define como *chart* al contenido de `source/deploy/chart`.

An.9 Cliente

Código fuente del cliente que envía datos a la *API CoAP*.

Se encuentra en `source/src/client`.

An.10 CronJobs para el envío periódico de medidas

Manifiestos de diferentes *CronJobs* de *Kubernetes* que hacen que se envíen medidas de forma periódica a la API.

Se encuentran en `source/misc/k8s-sender`.