eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

DESPLIEGUE DE UNA ARQUITECTURA DE EDGE COMPUTING BASADA EN KUBERNETES APLICADA A INDUSTRIA 4.0

Estudiante de Francisco Calvo, Cristina
Director Jacob Taquet, Eduardo
Departamento Ingeniería de comunicaciones
Curso académico 2020-2021

Bilbao, 21 de septiembre de 2020

RESUMEN:

El presente proyecto pretende diseñar y desplegar una arquitectura de *edge computing* para industria 4.0 basada en Kubernetes (K8s), el sistema de orquestación de contenedores más adoptado hoy en día por las empresas tecnológicas. Para ello, se analizarán las necesidades de los entornos de *edge computing*, así como las de la industria 4.0. Una vez hecho eso, se analizarán las distribuciones de K8s que mejor se adapten a esas necesidades. Se diseñará y realizará una metodología para evaluar la utilización de recursos y las prestaciones más adecuadas, seleccionando la distribución más idónea al despliegue. La solución propuesta tratará de aportar nuevos servicios de gestión de planta a las empresas que están evolucionando hacia la emergente industria 4.0. De este modo, podrán tomar decisiones de forma más rápida sobre la gestión de los dispositivos y equipos distribuidos en sus plantas de fabricación. Se validará por medio del despliegue de una Prueba de Concepto (PoC) de gemelo digital y lectura de sensores.

ABSTRACT:

This project aims to design and deploy an edge computing architecture for the 4.0 industry based on Kubernetes (K8s), currently the most adopted container orchestration system in all technology companies. In order to attain that, the project will first present a study on the needs of the edge computing environment, as well as the needs of the 4.0 industry. Once those needs are defined, an analysis of the different K8s distributions that best fit those needs will be done, as well as a methodology design and implementation to evaluate the use of resources and performance of each one of them. Based on that analysis, the most suitable alternative will be selected for the final deployment. The proposed solution will bring new management services to companies that are evolving towards the emerging 4.0 industry. These new services will help them make quicker decisions about the management of the devices and equipment distributed in the manufacturing plants. This will be validated through the deployment of a *Proof of Concept* (PoC), based on digital twins and sensor readings.

LABURPENA:

Proiektu honek 4.0 industriarako *edge computing* arkitektura bat diseinatzea eta zabaltzea du helburu, Kubernetes-en (K8s) oinarriturikoa, enpresa teknologikoek gehien erabiltzen duten kontainer orkestrazio-sistema baita. Horretarako, *edge computing* inguruneen beharrak aztertuko dira, baita 4.0 industriarenak ere. Hori egin ondoren, behar horietara hobekien egokitzen diren K8s-en banaketak aztertuko dira. Baliabideen erabilera eta prestazio egokienak ebaluatzeko metodologia bat diseinatu eta garatuko da, proiektu honen arkitekturarako banaketa egokiena aukeratuz. Zabalduko den aplikazioak plantaren kudeaketa-zerbitzu berriak ekarriko dizkie garatzen ari den 4.0 industriarantz eboluzionatzen duten enpresei. Horrela, beren fabrikazio-instalazioetan banatutako gailu eta ekipoen kudeaketari buruzko erabakiak azkarrago hartu ahal izango dituzte. Hau kontzeptu-proba (PoC) baten bidez baliozkotuko da, biki digitaletan eta sentsoreen irakurketan oinarriturikoa.

PALABRAS CLAVE:
computación en el borde, industria 4.0, Kubernetes, KubeEdge, microservicios, gemelo digital

KEY WORDS:
edge computing, industry 4.0, Kubernetes, KubeEdge, microservices, digital twin

GAKO-HITZAK:
ertz konputazioa, 4.0 industria, Kubernetes, KubeEdge, mikrozerbitzuak, biki digitala

Listado de acrónimos

API	*Application Programming Interface*
BD	Base de Datos
CI/CD	*Continuous Integration, Continuous Delivery and Deployment*
CLI	*Command Line Interface*
CNCF	*Cloud Native Computing Foundation*
CNI	*Container Network Interface*
CRD	*Custom Resource Definition*
DNS	*Domain Name System*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IA	Inteligencia Artificial
IaaS	*Infrastructure as a Service*
IIoT	*Industrial* IoT
IOPS	*I/O Operations Per Second*
IoT	*Internet of Things*
IP	*Internet Protocol*
K8s	Kubernetes
KPI	*Key Performance Indicator*
MEC	*Multi-access Edge Computing*
MQTT	*Message Queuing Telemetry Transport*
OPC	*Open Platform Communications*
PaaS	*Platform as a Service*
PoC	*Proof of Concept*
QUIC	*Quick UDP Internet Connections*
RSS	*Resident Set Size*
SaaS	*Software as a Service*
TCP	*Transmission Control Protocol*
TSN	*Time Sensitive Network*
UDP	*User Datagram Protocol*
VM	*Virtual Machine*

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

La rápida evolución de la tecnología y el uso cada vez más extendido de ella en todos los ámbitos de la sociedad brinda oportunidades para adaptarnos a nuevas situaciones de una forma diferente. Los dispositivos son cada vez más pequeños, potentes y autónomos, al mismo tiempo que crece la demanda por la innovación. Un buen reflejo de ello son las empresas industriales, que buscan una continua incorporación de estas emergentes tecnologías para mejorar la productividad y el rendimiento de sus procesos de fabricación.

De esa creciente exigencia por nuevas soluciones tecnológicas surge la industria 4.0 como una cuarta revolución industrial, centrada en un incremento de la productividad impulsado por la seguridad, la automatización y una toma de decisiones más rápida e inteligente que nunca. En Euskadi, sin ir más lejos, la transformación digital de las empresas industriales está promovida por el Gobierno Vasco a través del Basque Digital Innovation Hub (BDIH), una red de colaboración pública-privada que proporciona a las empresas industriales las capacidades tecnológicas necesarias para hacer frente a los desafíos de la industria 4.0 [1].

Los principales avances que acompañan a la industria 4.0 son el Internet de las Cosas (IoT, *Internet of Things*), la inteligencia artificial (IA) y el procesado masivo de datos o *big data*. Estos avances van acompañados de una mayor carga de trabajo y capacidad de procesamiento, potenciadas por la virtualización, los microservicios y el *cloud computing* o computación en la nube. Sin embargo, junto con el crecimiento exponencial del uso del IoT y la IA también ha aumentado el tráfico generado por los dispositivos finales, que atraviesa toda la red hasta llegar a la nube. Trasladar todo el procesamiento a la nube implica un coste en el ancho de banda de las redes tanto públicas como privadas, así como un incremento en los tiempos de respuesta. Esto puede llegar a generar cuellos de botella en los paradigmas de la computación en la nube, por lo que resulta necesario considerar nuevas arquitecturas y estrategias que den solución a este problema.

Una de estas nuevas arquitecturas a tener en cuenta es el *edge computing* o computación en el borde, basado en acercar el procesamiento de los datos a un punto más cercano al origen de dichos datos. Esto permite reducir la latencia y los tiempos de respuesta, así como aumentar la privacidad de los datos y reducir el tráfico hacia el *core* de la red, abriendo nuevas puertas al procesamiento en tiempo real.

Según una encuesta llevada a cabo por la CNCF (*Cloud Native Computing Foundation*) [2], actualmente el 58% de las empresas industriales y tecnológicas hacen uso de algún tipo de procesado en el borde. Los casos más populares de uso son el de IoT para fabricación, la telefonía móvil y el procesado de imagen.

En cuanto al crecimiento del mercado, según previsiones realizadas por diferentes empresas de investigación y consultorías de mercado [3] [4], se espera que el mercado del *edge computing* tenga entre 2021 y 2028 una tasa de crecimiento anual compuesto de alrededor del 30%. Con un valor de más de 3 mil millones de euros en 2020, se espera que el mercado alcance un valor de 25 mil millones de euros en 2028, según las estimaciones más conservadoras. Esto demuestra el interés cada vez mayor de las empresas y la necesidad de nuevos planteamientos que permitan extender esas capacidades de procesado de la nube al borde de la red.

2. Contexto

En este apartado se tratará de contextualizar el trabajo, de tal forma que sirva como referencia para una mejor comprensión del resto del documento.

2.1. Industria 4.0

La industria 4.0 es un término que surge de los desarrollos tecnológicos de la última década que han impulsado la cuarta revolución industrial. Se caracteriza principalmente por la automatización de procesos de producción, con control descentralizado y en tiempo real, y una alta conectividad entre dispositivos. Esto ha permitido a las empresas industriales obtener una producción en masa personalizada y flexible.

2.1.1. Conceptos fundamentales

En este nuevo paradigma industrial toman relevancia conceptos como el Internet de las Cosas (IoT, *Internet of Things*), la inteligencia artificial (IA) y el procesado masivo de datos o *big data*. Todos ellos suelen ser conceptos interrelacionados y aplicados en conjunto.

El IoT es el nombre que se da al sistema que forman los miles de millones de dispositivos que están conectados a internet y que son capaces de recoger e intercambiar datos entre ellos. Es un término que hoy en día engloba casi todos los ámbitos de la sociedad, desde el campo de la salud, con dispositivos médicos y pulseras de actividad, pasando por las bombillas, termostatos y demás dispositivos que encontramos en los hogares inteligentes, hasta sensores de temperatura, presión, vibración, etc. que rodean una máquina de una línea de producción industrial. Cuando se habla de un IoT aplicado de forma específica a la industria se suele hablar de IIoT (*Industrial* IoT).

La IA es un término que hace referencia a una máquina que es capaz de tomar decisiones, imitando el razonamiento humano. Para hacer esto posible se aplican técnicas de aprendizaje basadas en datos previamente conocidos y sus predicciones, también llamadas técnicas de *machine learning*.

El concepto de *big data* hace referencia a grandes volúmenes de datos complejos y que crecen de forma exponencial, de los cuales se extrae información con el objetivo de observar patrones. Está estrechamente relacionado con el enorme conjunto de datos que genera el IoT y que, generalmente, se procesa con técnicas de *machine learning*.

Adicionalmente, otro concepto que ha tenido siempre relación con la industria en general, pero que toma especial relevancia en la industria 4.0 es el mantenimiento preventivo. Se basa en la posibilidad de predecir tendencias y patrones de comportamiento mediante el aprendizaje automático basado en datos históricos. Esto permite anticipar fallas en los sistemas y procesos, de tal forma que se puedan tomar decisiones de mantenimiento de estos de una forma más eficiente y disminuyendo el

tiempo de inactividad [5]. Esto, a su vez, mejora la productividad y la calidad de los procesos. Los datos recopilados de los múltiples sensores en entornos de la industria 4.0, así como nuevas tecnologías como el *machine learning* o la IA, ofrecen nuevas oportunidades para soluciones de predicción de la vida restante de un activo.

2.1.2. *Smart factory*

El esquema general de una *smart factory* o fábrica inteligente está formado por celdas de fabricación flexible o líneas de producción que internamente tienen un bus de campo. Este bus de campo comunica en tiempo real los dispositivos con los sistemas de automatización y suele ser, generalmente, Profinet o TSN (*Time Sensitive Network*), en los casos más modernos. Ambas tecnologías están basadas en Ethernet y permiten el control de los datos y la toma de decisiones en tiempo real. Estas decisiones se toman de forma local, de manera que el dato no sale fuera. Los únicos datos que se envían fuera de la red de la empresa son datos estadísticos, datos de medida o alarmas que no se han podido solucionar de forma local. Para ello, se suele incluir en el esquema un *gateway* que interconecta la red Profinet con la red IP (*Internet Protocol*) de la empresa, a través de la cual encaminar los datos hacia el exterior. En la Figura 1 se muestra de forma gráfica este esquema.

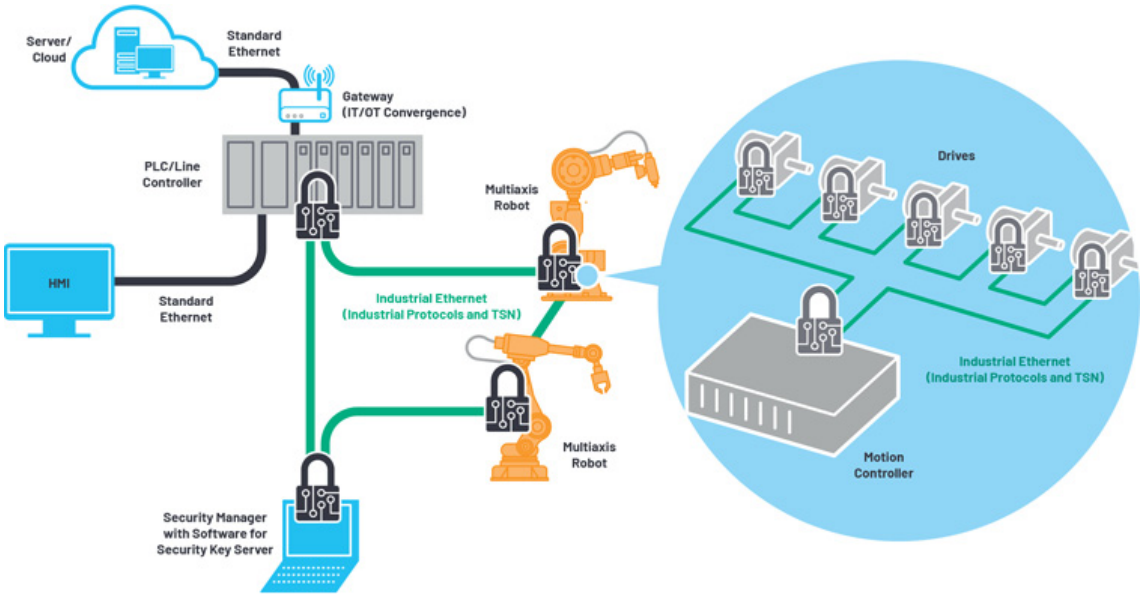

Figura 1. Esquema de red de una *smart factory* [6].

Sin embargo, en aquellos casos en los que no existen requerimientos o garantías de tiempo, ni haya una criticidad importante, resulta interesante considerar una red completamente IP. En este escenario, todos los datos irían directamente contra un servidor donde se procesarían y se realizaría la toma de decisiones. Este servidor externo puede tener varias localizaciones, dependiendo de las necesidades del procesamiento

de cada tipo de dato. Generalmente, se consideran 3 localizaciones: el *cloud*, el *edge* y las premisas de la propia fábrica, comúnmente denominado *local edge*.

2.1.3. Gemelo digital

Cuando se habla de *smart factories* e industria 4.0, otro de los términos en auge es el de gemelo digital. Este concepto surge de la idea de crear un modelo virtual de la fábrica y sus dispositivos, un gemelo digital, sobre el que simular y optimizar el proceso de toma de decisiones, facilitando todas las operaciones de la fábrica y monitorizando su estado en todo momento.

En resumen, un gemelo digital se trata de una representación virtual de un objeto o un sistema que abarca todo su ciclo de vida y se actualiza en base a datos que recibe en tiempo real. Estos datos son, generalmente, datos que recogen múltiples sensores que rodean a un dispositivo, ya sea un coche autónomo, un robot, una máquina industrial, etc. Esto permite no solo monitorizar el rendimiento de los dispositivos, sino también predecir el comportamiento que tendrán en el futuro. El análisis de los datos de los sensores, junto con la aplicación de algoritmos de *machine learning*, permite hacer esas predicciones para ofrecer un mantenimiento predictivo y preventivo. En la Figura 2 se muestra el esquema de funcionamiento de un gemelo digital.

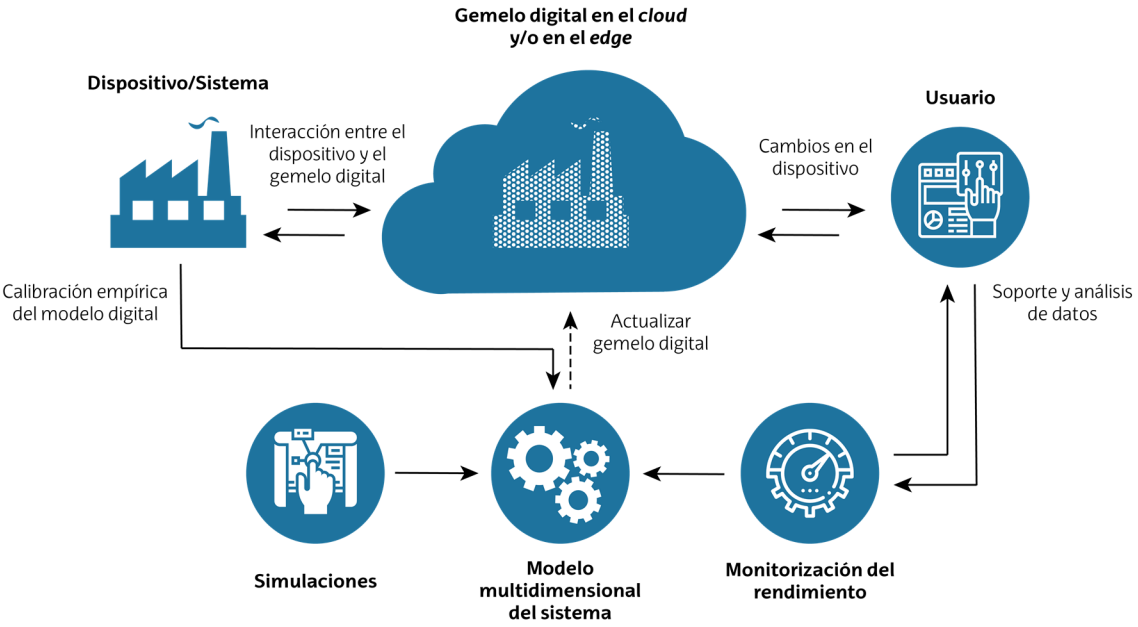

Figura 2. Esquema de funcionamiento de un gemelo digital [7].

Gestamp, empresa española de diseño, desarrollo y fabricación de componentes automovilísticos, desarrolló a finales de 2020 varios prototipos de gemelo digital en sus plantas de Barcelona y, con la ayuda de la Universidad del País Vasco (UPV/EHU), Euskaltel está implantando otro en la fábrica de Abadiño (Bizkaia). En esos prototipos

hacían uso del 5G y tecnologías MEC (*Multi-access Edge Computing*) para optimizar la comunicación entre la planta, los dispositivos y el gemelo digital, minimizando la latencia [8] [9].

2.1.4. MQTT

MQTT (*Message Queuing Telemetry Transport*) es el protocolo de comunicación por excelencia en entornos IoT. Es un protocolo diseñado para redes en las que el ancho de banda es limitado y para dispositivos IoT con necesidades de baja latencia. Estas dos características lo convierten en un protocolo ideal para las comunicaciones máquina a máquina o M2M (*machine to machine*). Además, su implementación es muy ligera y sencilla, lo cual permite ahorrar batería a los dispositivos.

Su arquitectura se compone de un servidor, llamado MQTT *broker*, y varios clientes, siguiendo un modelo de publicación y suscripción. Un único *broker* puede soportar miles de clientes conectados de forma simultánea. Es el encargado de recibir los mensajes, filtrarlos y mostrárselos a aquellos clientes que están suscritos. También se encarga de la autenticación y autorización de los clientes. En esencia, se encarga de encaminar los mensajes entre clientes.

La comunicación se basa en la definición de temas sobre los cuales los clientes pueden publicar mensajes, o suscribirse a ellos. Una de sus principales ventajas es que permite almacenar mensajes para cuando la conexión con cierto cliente se ve interrumpida. Una vez el cliente suscriptor vuelve a conectarse, se le hace llegar el mensaje almacenado. En la Figura 3 se muestra un esquema general de su funcionamiento.

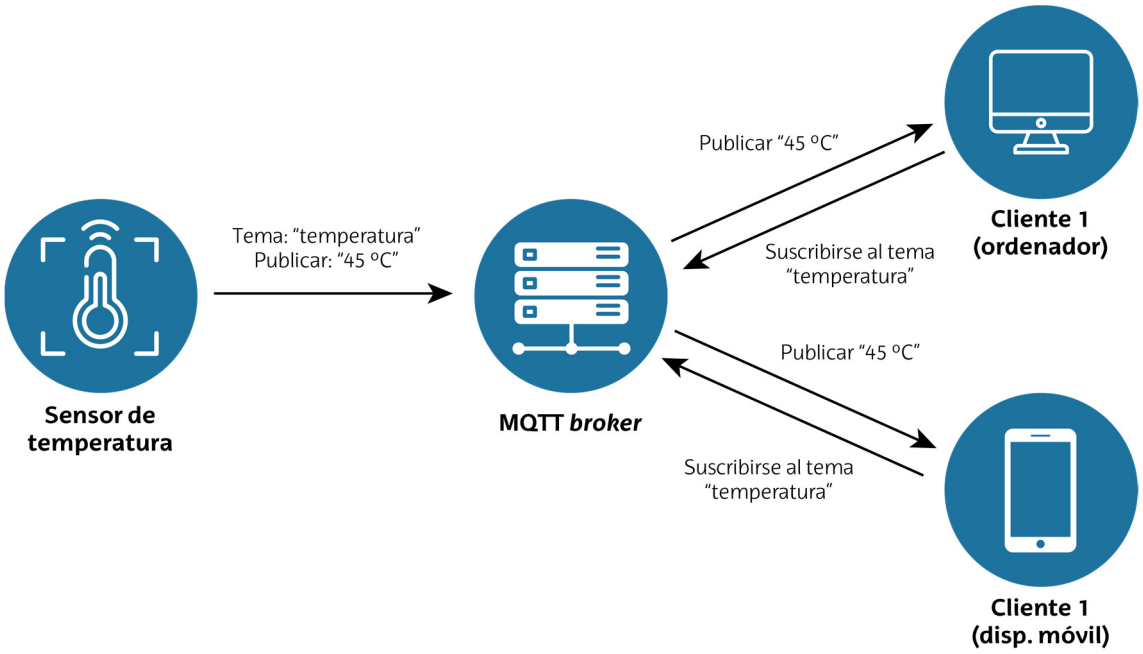

Figura 3. Esquema general de funcionamiento de MQTT [10].

2.2. *Cloud computing* y *edge computing*

El *cloud computing* o computación en la nube es un modelo de servicios basado en una gran cantidad de servidores centralizados alojados en centros de datos. El usuario paga por acceder a servicios que están alojados en dichos servidores centralizados. Esos servicios se suelen categorizar en 3 tipos: acceso a plataformas sobre las que poder desplegar software y aplicaciones personalizadas, pero sin control de la infraestructura (PaaS, *Platform as a Service*); acceso a software específico (SaaS, *Software as a Service*); y acceso a infraestructura con control sobre el sistema operativo, las aplicaciones y el almacenamiento (IaaS, *Infraestructure as a Service*).

Las principales ventajas del *cloud computing* son su gran escalabilidad y flexibilidad del despliegue, alta disponibilidad y menores costes de equipamiento y mantenimiento. Sin embargo, esta arquitectura resulta engorrosa para procesos que requieren cálculos intensivos y en tiempo real, siendo la latencia el principal problema. Además, en este modelo, características como el ancho de banda o la latencia suelen tener un alto coste.

El modelo de *edge computing* pretende descentralizar el almacenamiento y procesado de los datos del *cloud*, distribuyendo la infraestructura en localizaciones más cercanas a los dispositivos que generan los datos. Dado que los datos se procesan de forma local, se evita tener que transferir grandes cantidades de datos en tiempo real entre una localización remota y el *cloud*. Esto permite aumentar el rendimiento del procesamiento y reducir los costes operativos. A su vez, se ve reducida la latencia, así como el ancho de banda ocupado en el envío de tráfico al *cloud*. Además, el procesado local aporta un nivel adicional de seguridad, ya que permite analizar datos confidenciales dentro de una red privada, protegiendo dichos datos. Ese mayor rendimiento, mayor seguridad y menores costes de procesado hacen del *edge computing* un modelo cada vez más atractivo para las empresas industriales. Es especialmente útil en entornos donde se despliegan muchos sensores IoT, cuyos datos recogidos requieren de un procesamiento en tiempo real, como podría ser el caso de los vehículos autónomos. Sin embargo, también resulta beneficioso en otros entornos, como es el caso de los servicios de *streaming* con cacheo de contenido en el *edge*.

Aunque los conceptos de *cloud computing* y *edge computing* se suelen presentar como opuestos, generalmente se suele optar por soluciones híbridas. Al fin y al cabo, el *edge computing* es una extensión de la arquitectura del *cloud computing* que resuelve algunas de sus limitaciones y reduce la dependencia en el *cloud*, pero sin eliminarla por completo [11].

2.3. Evolución de la virtualización

Cuando se habla de la evolución de la virtualización se suele hacer mención a 3 fases de la arquitectura de los despliegues de aplicaciones: la tradicional, la basada en máquinas virtuales y la basada en contenedores. En la Figura 4 se resumen de forma gráfica las capas y componentes de esas 3 arquitecturas.

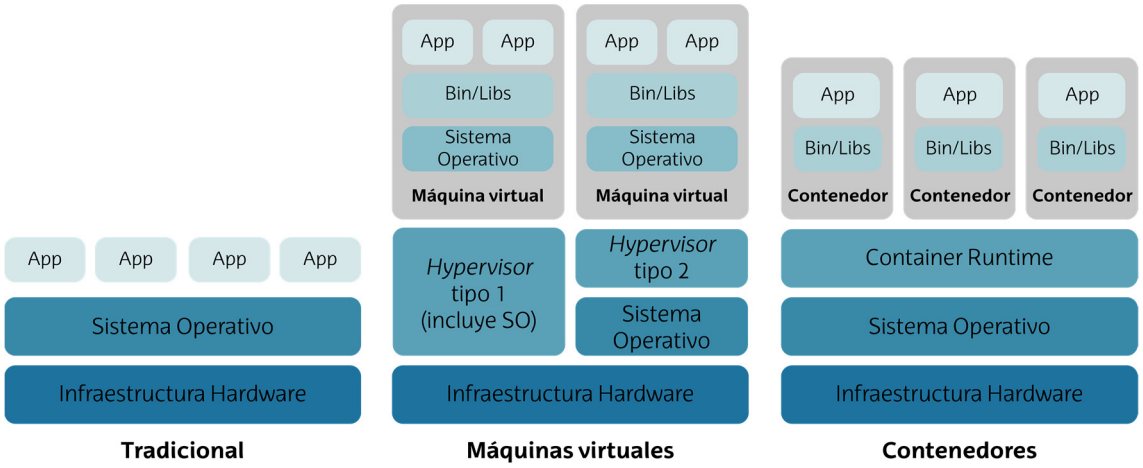

Figura 4. Evolución de la arquitectura de despliegue de aplicaciones.

En un despliegue tradicional las aplicaciones se ejecutan sobre servidores físicos, sin restricciones de uso de los recursos para cada una de ellas. Esto implica que, si en el servidor hay más de una aplicación ejecutándose, una de ellas podría hacer un mayor uso de los recursos, dejando al resto sin los suficientes. Si se llevase este modelo al extremo, se ejecutaría una única aplicación por cada servidor y no existirían conflictos entre aplicaciones, pero esto conllevaría un uso poco eficiente de los recursos disponibles.

La virtualización aparece como solución a la problemática del despliegue tradicional, pudiendo ejecutar múltiples instancias o máquinas virtuales (VM, *Virtual Machine*) sobre una misma infraestructura hardware. Las aplicaciones se pueden aislar entre VMs, pudiendo asignar a cada VM una serie de recursos virtualizados de uso exclusivo, mapeados al hardware físico. El software encargado de realizar ese aislamiento y asignación de recursos es el *hypervisor*. Existen varios tipos de *hypervisor*, uno al que se refiere como de tipo 1 en el que el propio *hypervisor* actúa como un sistema operativo ligero y otro al que se llama de tipo 2 en el que el *hypervisor* se ejecuta como una capa de software sobre el sistema operativo. Cada VM contiene su propio sistema operativo que se ejecuta sobre el hardware virtualizado, permitiendo ejecutar sobre el mismo equipo físico múltiples sistemas operativos de forma paralela. No obstante, si el sistema operativo de todas las VMs a desplegar es el mismo, resulta poco eficiente ejecutar para cada una de ellas la capa de sistema operativo.

Como solución a eso aparecen los contenedores. Son similares a las VMs, pero comparten el sistema operativo del equipo anfitrión o *host*. Esto aporta ligereza a los despliegues, pero también supone una limitación. Sin embargo, los contenedores no están limitados a despliegues sobre equipos físicos, sino que pueden desplegarse dentro de las propias VMs. Esto permitiría, por ejemplo, tener sobre un mismo equipo físico, una VM por cada sistema operativo deseado y sobre cada uno de ellos desplegar múltiples contenedores con las aplicaciones. De forma similar al *hypervisor* en el esquema de las máquinas virtuales, para ejecutar los contenedores es necesario tener por debajo un gestor de contenedores. Existen múltiples gestores de contenedores en la actualidad, siendo los más extendidos Docker [12], containerd [13], CRI-O [14] y LXD [15].

La ligereza de los contenedores hace que su despliegue sea más rápido y que se integren fácilmente en prácticas CI/CD (*Continuous Integration, Continuous Delivery and Deployment*). Estas prácticas hacen posibles metodologías más ágiles de desarrollo de software, automatizando los despliegues de nuevas versiones del código.

El uso de contenedores también hace más fácil la división de las aplicaciones en piezas más pequeñas e independientes, siguiendo una arquitectura de microservicios. En este esquema, cada servicio se desarrolla y despliega de forma independiente a los demás y se comunican entre ellos mediante interfaces definidas, llamadas API (*Application Programming Interface*), que permiten abstraer la lógica y el código interno de cada servicio. Para gestionar el despliegue de estos microservicios suele ser necesario implementar sistemas de orquestación, como Kubernetes.

2.3.1. Kubernetes

Kubernetes (K8s) es una plataforma de código abierto que facilita la orquestación de contenedores para automatizar los despliegues, el escalado y la gestión de aplicaciones [16]. Tiene su origen en Google, que liberó su código en 2014, y está alojado actualmente en la CNCF (*Cloud Native Computing Foundation*). Esta fundación sin ánimo de lucro se creó con el objetivo de impulsar nuevas tecnologías de computación en la nube, creando un ecosistema de proyectos de código abierto, impulsando y adoptando los más innovadores y con mayor futuro. Los proyectos que adopta el CNCF se categorizan según su nivel de madurez como proyectos *sandbox*, *incubating* o *graduated*, de menor a mayor madurez, respectivamente. El nivel de madurez de cada proyecto se define en base a su adopción entre los usuarios finales, la cantidad de actualizaciones y revisiones del código, y su capacidad de despliegue en un entorno de producción. Kubernetes se trata de un proyecto *graduated* [17].

Como ya se ha explicado en apartados anteriores, los contenedores permiten ejecutar de forma ligera todo tipo de aplicaciones en una gran variedad de entornos. Necesitan menos capacidad y recursos que una máquina virtual, además de tener un tiempo de arranque también menor. Sin embargo, resulta necesario un nivel de orquestación que realice un seguimiento de todos los contenedores desplegados y los planifique, tanto en el tiempo como en la utilización de recursos. Kubernetes se encarga de esa orquestación, facilitando la configuración, el despliegue, la gestión y la monitorización de las aplicaciones y los contenedores sobre los que estas corren, incluso en despliegues a gran escala. Además, ayuda en la gestión del ciclo de vida tanto de los contenedores como de las aplicaciones, así como en la provisión de alta disponibilidad y balanceo de carga. Permite gestionar cuántos contenedores necesitan estar desplegados, qué servicios corre cada uno y los recursos asignados. Esto hace que exista un mayor nivel de coordinación, combinando los contenedores individuales como una entidad global.

Para gestionar la orquestación de los contenedores, Kubernetes organiza los nodos sobre los que se ejecutarán las aplicaciones en grupos llamados *clusters*. Estos nodos pueden ser ordenadores físicos o máquinas virtuales. Esto permite a Kubernetes abstraer la agrupación de nodos en una única entidad, pudiendo programar y ejecutar contenedores en un grupo de nodos, sin tener que asociar cada contenedor a un nodo específico.

Un *cluster* Kubernetes se compone de un nodo de control, generalmente llamado *master node*, y uno o más nodos de cómputo o *worker nodes*. El nodo de control se encarga de supervisar y mantener el estado deseado del *cluster*, previamente definido (por ejemplo, qué aplicaciones o cuántas réplicas de cada una se requieren). Los nodos de cómputo son los que se encargan de albergar los contenedores sobre los que se ejecutan las aplicaciones, en función de los recursos asignados por el nodo de control.

De la misma forma que abstrae un conjunto de nodos en un *cluster*, Kubernetes basa su arquitectura en abstracciones de los diferentes componentes de los sistemas para poder gestionar de forma genérica los microservicios a desplegar y realizar operaciones sobre ellos. Los principales objetos que componen la arquitectura son los siguientes:

- **Pod**: representa la unidad más pequeña de computación a desplegar en Kubernetes. Un *pod* se puede componer tanto de un único contenedor como de varios contenedores con un almacenamiento y recursos de red compartidos. Esa compartición de recursos permite que los procesos de un mismo *pod* puedan comunicarse entre ellos mediante *sockets* u otras técnicas de comunicación entre procesos. Son entidades relativamente efímeras, con un tiempo de vida generalmente reducido.
- **Service**: permite abstraer el acceso a una aplicación que se ejecuta en una serie de *pods*, como un único servicio disponible en la red de *pods*. En el momento de

su creación, a cada *pod* se le asigna una dirección IP propia, pero dado que los *pods* son efímeros y pueden crearse y destruirse de forma dinámica en el *cluster*, es necesario una forma de hacer referencia a aquellos que actualmente están disponibles y ejecutando una aplicación en concreto, sin hacer uso de las direcciones IP variables. La abstracción de servicios permite asociar a todos los *pods* que ejecuten una misma aplicación un nombre de DNS (*Domain Name System*) mediante el cual poder referenciarlos de forma única, realizando balanceo de carga entre todos los *pods* que ejecutan el mismo servicio.

- **Volume**: representa un directorio que es accesible por todos los contenedores dentro de un *pod*. Puede ser de tipo efímero, que se destruye al mismo tiempo que el *pod*, o persistente, cuyo ciclo de vida no depende del ciclo de vida del *pod*.
- **Controller**: controlador que observa el estado del *cluster* y realiza cambios sobre él. Gestiona los recursos para poder replicar o redesplegar *pods*, de tal forma que el *cluster* esté siempre en un estado deseado. Para describir ese estado deseado se utilizan abstracciones de la carga de trabajo. Existen 3 tipos básicos de descriptores de cargas de trabajo: *Deployments*, para despliegues de aplicaciones sin estado, *StatefulSets*, para despliegues de aplicaciones que requieren mantener un estado independiente y persistente, y *DaemonSets*, para gestionar el despliegue de una copia del mismo *pod* en cada uno de los nodos del *cluster* [18].

Para añadir soporte a abstracciones personalizables, Kubernetes dispone de extensiones de su API llamadas CRDs (*Custom Resource Definitions*). Estas extensiones permiten definir nuevos objetos de la API, con atributos personalizables. A su vez, hacen de Kubernetes una plataforma más modular.

En la Figura 5 se representa la arquitectura general de un *cluster* de Kubernetes, formado por un nodo *master* de control y 2 nodos *worker*. El nodo *master* contiene los 4 componentes del plano de control:

- Un servidor API (**kube-apiserver**) que expone la API de Kubernetes para que el administrador del *cluster* pueda ejecutar acciones como crear despliegues, obtener la lista de nodos desplegados, etc. Generalmente, estas acciones se ejecutan mediante una herramienta CLI (*Command Line Interface*), llamada **kubectl**.
- Un planificador (**kube-scheduler**) que determina sobre qué nodo se va a desplegar cada nuevo *pod* creado.
- Un gestor de procesos de controladores (**kube-controller-manager**). Los controladores a gestionar se encargan de observar en todo momento el estado de los nodos, los servicios, etc.
- Almacenamiento (**etcd**) de todos los datos del *cluster* como pares clave-valor.

En los nodos *worker*, en cambio, es necesario disponer, por un lado, de un motor de contenedores, que puede ser Docker, containerd o CRI-O, y, por otro lado, dos componentes específicos de Kubernetes:

— **kubelet**: se encarga de observar los contenedores desplegados con Kubernetes, asegurándose de que se están ejecutando de forma adecuada. Dicho de otra forma, gestiona el ciclo de vida de los *pods*.
— **kube-proxy**: se encarga de gestionar las reglas de red para la comunicación con los *pods*, ya sea desde dentro del *cluster* o desde el exterior [19].

Adicionalmente, Kubernetes utiliza *plugins* de red externos para la comunicación entre contenedores, llamados CNI (*Container Network Interface*). Estos *plugins* definen las librerías y especificaciones para configurar las interfaces de red de los contenedores. Los CNI más utilizados junto a Kubernetes son Flannel y Calico.

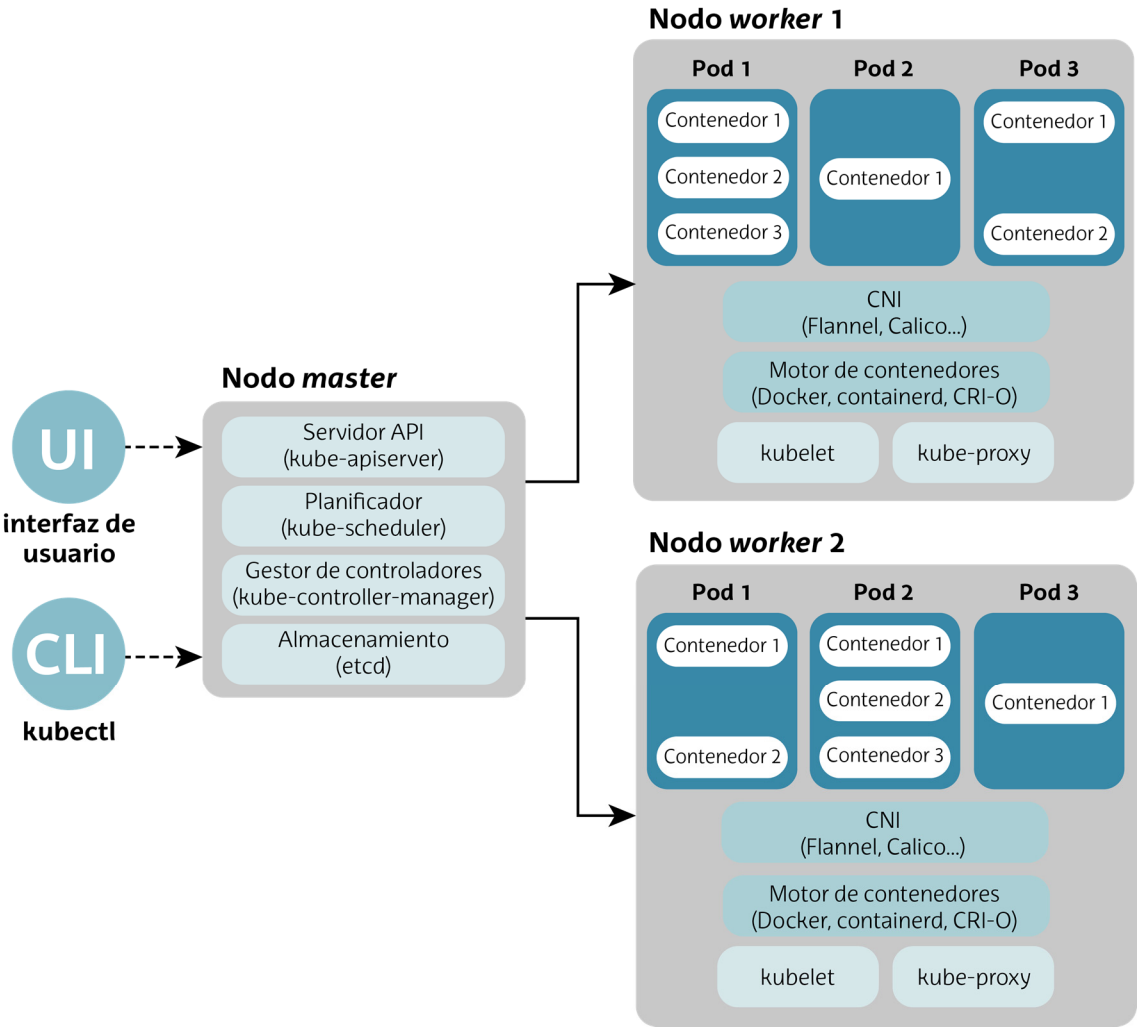

Figura 5. Arquitectura general de un *cluster* Kubernetes.

La mayor ventaja de Kubernetes es su alto nivel de adopción y uso por parte de las empresas tecnológicas de todo el mundo, lo cual lo convierten en la plataforma orquestadora líder del mercado. Es uno de los proyectos de código abierto con mayor velocidad de desarrollo hasta la fecha [20]. Está en desarrollo y crecimiento constante, lo cual ofrece la seguridad de ser una plataforma con soporte de cara al futuro. Además, se adapta a todo tipo de soluciones, independientemente del nivel de escala de la solución, del proveedor de servicios o de la propia infraestructura subyacente.

A pesar de ser una de las soluciones más extendidas, tiene una curva de aprendizaje compleja. Asimismo, la transición a Kubernetes suele conllevar tiempo y costes que muchas empresas, principalmente pequeñas, no están todavía dispuestas a afrontar.

Sin embargo, otra ventaja de Kubernetes es el gran ecosistema de herramientas al que pertenece. Como ya se ha mencionado, es uno de los proyectos alojados por el CNCF, que, a su vez, dispone de otras muchas herramientas que se complementan. Además, muchas de esas herramientas se han desarrollado para su uso específico junto a Kubernetes. El uso de esas herramientas complementarias permite reducir su complejidad de uso, así como hacer el despliegue más eficiente y personalizable. Ejemplo de ello son herramientas como el gestor de paquetes y aplicaciones Helm o el sistema de monitorización Prometheus.

Para los casos de uso de *edge computing*, Kubernetes también presenta otra desventaja en la forma de gestionar los nodos cuya conexión se pierde. Cuando un nodo se desconecta, Kubernetes pasa ese nodo a un estado desconocido, en el que elimina de la lista de servicios los *pods* desplegados en el nodo desconectado para redesplegarlos en otros nodos que sí están disponibles. En consecuencia, cuando los nodos cuya conexión se ha perdido vuelven a conectarse, los *pods* se eliminan de estos y el servicio deja de prestarse desde ese nodo. Esto representa un inconveniente en casos de *edge computing*, donde la pérdida de conectividad entre el *cloud* y el *edge* es habitual.

3. Objetivos y alcance del trabajo

La principal finalidad del proyecto es desplegar en un entorno privado una solución de *edge computing* con aplicación en la industria 4.0, basado en un *cluster* de Kubernetes. La solución propuesta deberá poder validarse mediante el despliegue de una aplicación que se aplique a casos de uso específicos de *edge computing*, como son el gemelo digital y la lectura de medidas de sensores IIoT. Teniendo dicha finalidad en cuenta, se deberán abordar los siguientes objetivos específicos:

- Estudiar las especificaciones y requerimientos de un despliegue de *edge computing*.
- Estudiar las necesidades de la industria 4.0.
- Analizar las distribuciones de Kubernetes que cumplan con los requerimientos de un despliegue de *edge computing* y den solución a las necesidades de la industria 4.0.
- Proponer una metodología para medir el uso de recursos y el rendimiento de las distribuciones analizadas. También será necesario realizar las medidas de rendimiento de las alternativas de despliegue y evaluar los casos de uso en los que resulta interesante su aplicación.
- Diseñar e implementar la arquitectura del sistema, en base a las alternativas seleccionadas.
- Desplegar una prueba de concepto (PoC, *Proof of Concept*) de un gemelo digital y lectura de IIoT que sirva para evaluar la solución propuesta.

4. Beneficios que aporta el trabajo

En el presente Trabajo de Fin de Máster se propone una solución que unifica los beneficios del uso del *edge computing* y Kubernetes para las empresas industriales 4.0. En este apartado se describirán esos beneficios conjuntos.

4.1. Beneficios técnicos

Entre los beneficios técnicos destaca la posibilidad para las empresas de monitorizar en tiempo real sus plantas de producción. Esto es, en gran parte, debido a que el uso del *edge computing*, dado a su proximidad al origen de los datos, minimiza la latencia, haciendo posible aplicaciones en tiempo real. Además, Kubernetes aporta gran escalabilidad al sistema, permitiendo monitorizar un mayor número de equipos. El uso de tecnologías de gemelos digitales también hace posible que los administradores de la planta monitoricen y controlen el rendimiento de todo el sistema de forma remota y en tiempo real.

Como ya se ha mencionado, Kubernetes es una plataforma que se usa desde hace varios años de forma extensa en el entorno de *cloud computing*, pero también ofrece grandes ventajas para los entornos de características exigentes de *edge computing*. Las nuevas distribuciones de Kubernetes específicamente diseñadas para estos entornos añaden una capa adicional de ventajas. Por un lado, es posible optimizar el uso de los recursos computaciones del *edge*. Estos recursos son, generalmente, limitados y es necesario utilizarlos de forma eficaz y preservando las funcionalidades críticas que suelen requerir las aplicaciones desplegadas en este entorno. Las distribuciones que se analizan en este proyecto están específicamente orientadas a un uso mínimo de los recursos de los sistemas. Además, la selección de la plataforma de despliegue realizada en el presente proyecto se ha basado en el estudio sistemático de los recursos utilizados por las distribuciones, así como su rendimiento, siguiendo una metodología específicamente diseñada para este análisis.

Por otro lado, algunas de estas distribuciones de Kubernetes ofrecen la gestión y orquestación de nodos geográficamente dispersos, manteniendo la ventaja de escalabilidad de Kubernetes. Esto permite gestionar un mayor número de nodos, independientemente del estado de conectividad entre sedes o centros de datos.

4.2. Beneficios económicos

Aprovechando la gran cantidad de datos que generan los sensores IoT, el uso de aplicaciones de procesado en tiempo real de dichos datos hace posible identificar de forma proactiva cualquier problema del sistema presente y futuro. Esto permite a las empresas planificar con mayor precisión el mantenimiento de los equipos, mejorando

la eficiencia de sus líneas de producción. Esto, a su vez, se traduce en menores costes de mantenimiento de los equipos y, dado que minimiza el tiempo de inactividad de la planta por errores en los equipos, en una reducción de los costes operativos. Adicionalmente, el procesamiento de esos datos y su simulación virtual mediante gemelos digitales permiten tomar mejores y más rápidas decisiones, lo cual impacta de forma directa al coste operativo de las empresas.

El uso de plataformas de orquestación como Kubernetes permite reducir el coste de despliegue de aplicaciones, dado que permite hacer un uso más eficiente de los recursos. Además, permiten desplegar aplicaciones con mayor disponibilidad. Por ejemplo, si un nodo del sistema falla, Kubernetes despliega la aplicación de forma automática en otro nodo con capacidades similares, reduciendo el tiempo de inoperatividad. Esto también se traduce en mayores ganancias para las empresas. Dado que se trata de una plataforma gratuita de uso extendido y de código abierto, también se reducen costes de compra de licencias software.

5. Análisis del estado del arte

5.1. Distribuciones de Kubernetes para el *edge*

Cuando se despliega un entorno K8s, existen dos opciones: compilar desde código fuente y configurarlo de forma manual, a lo cual se le llama generalmente *vanilla* K8s; o instalar una versión pre-compilada y configurada de forma específica para ensalzar ciertas características del entorno en el que se quiere desplegar K8s. A esa versión pre-compilada se le llama comúnmente distribución.

La principal ventaja de utilizar distribuciones es simplificar el proceso de instalación y configuración. Sin embargo, resulta necesario seleccionar para cada entorno la distribución más adecuada. Cada distribución suele ofrecer diferentes características y ofrece diferentes soportes de componentes como el gestor de contenedores (Docker, containerd, CRI-O, LXC, CoreOS, etc.), el almacenamiento (MySQL, SQLite, Amazon Elastic Block Storage, etc.) o el CNI o gestor de red y comunicación entre contenedores (Flannel, Calico, etc.). En función de las características que soportan, las distribuciones serán más adecuadas para aplicaciones *multi-cluster* o *single-cluster*, entornos *cloud* o *edge*, etc.

En el caso concreto de las distribuciones diseñadas para su uso en *edge computing*, es importante que estas se adapten con facilidad a entornos IoT y se puedan desplegar en equipos con recursos más limitados.

5.1.1. KubeEdge

KubeEdge es una distribución de código abierto de Kubernetes con origen en Huawei, actualmente adoptada por el CNCF como *incubating project*. Está específicamente diseñada para su uso en un entorno *edge*, con un plano de control en el *cloud* y los nodos *worker* en el *edge*. Permite una instalación ligera en dispositivos con recursos computacionales y almacenamiento limitados, aunque requiere de la existencia de un *cluster* K8s previo. Esto implica que se debe instalar sobre una instalación base de K8s. Es una distribución apropiada para entornos IoT, que soporta la gestión de dispositivos y la comunicación con ellos mediante MQTT. En la Figura 6 se muestra la arquitectura en la que se basa [21].

En el nodo controlador del *cloud* se despliega un servicio llamado **CloudCore**, integrado por 3 componentes:

- **EdgeController**: controla la sincronización del servidor API de Kubernetes con las configuraciones, las aplicaciones y los nodos del *edge*. Dicho de otra forma, se encarga de gestionar los recursos del *edge* y su asignación a cada aplicación o servicio desplegado.

— **DeviceController**: gestiona los dispositivos que se conectan a los nodos *edge* y permite su monitorización desde el *cloud*. Hace la función de controlador del gemelo digital, realizando una copia del estado actual de los dispositivos y permitiendo hacer cambios sobre ese estado, que se propagan al propio dispositivo en el *edge*. Hace uso de los *Custom Resource Definitions* (CRDs) de Kubernetes para describir el estado o los metadatos de los dispositivos y sincronizar las actualizaciones entre *edge* y *cloud*.
— **CloudHub**: sincroniza la información del *cloud* con la información recibida de los nodos *edge*. Soporta los protocolos WebSocket y QUIC (*Quick UDP Internet Connections*) para la comunicación entre *cloud* y *edge*.

En cada uno de los nodos del *edge* se instalan un gestor de contenedores para desplegar los *pods*, un *broker* MQTT para la comunicación con los dispositivos externos, una base de datos para almacenar información del nodo y el servicio **EdgeCore** para controlar el estado del nodo *edge*. Este servicio, a su vez, se divide en varios componentes:

— **EdgeHub**: es la parte *edge* del canal de comunicación entre el *cloud* y el *edge*. Se comunica con el *CloudHub* del nodo controlador en el *cloud* para sincronizar ambas plataformas.
— **MetaManager**: se encarga de gestionar la persistencia de los metadatos locales. Es esencial para permitir la autonomía de los nodos *edge*, cuando pierden conectividad con el nodo controlador del *cloud*.
— **DeviceTwin**: es el encargado de gestionar el gemelo digital en el nodo *edge*. Realiza una abstracción de los dispositivos físicos y genera un mapeo del estado de cada dispositivo con la información del *cloud*, sincronizando el estado de los dispositivos entre *cloud*, los nodos *edge* y los dispositivos.
— **Edged**: gestiona el ciclo de vida de los objetos Kubernetes (*pod*, *volume*...), como haría el servicio *kubelet* en un despliegue tradicional de K8s, también llamado *vanilla* K8s. En la Figura 7 se muestra la diferencia entre la gestión del ciclo de vida de un *pod* con un despliegue de *vanilla* K8s y un despliegue con KubeEdge sobre K8s.
— **ServiceBus**: cliente HTTP que permite comunicarse con servicios que corren en otros nodos *edge*.
— **EventBus**: cliente MQTT que interactúa con el MQTT *broker* para suscribirse a los datos que los dispositivos envían y publicar los datos que modifican el estado de los dispositivos. En esta comunicación también toman parte los *Mappers* específicos a los protocolos con los que se conecta cada dispositivo (Bluetooth, Modbus, OPC...). Estos se encargan de mapear la información y adecuarla a un formato esperado por cada parte.

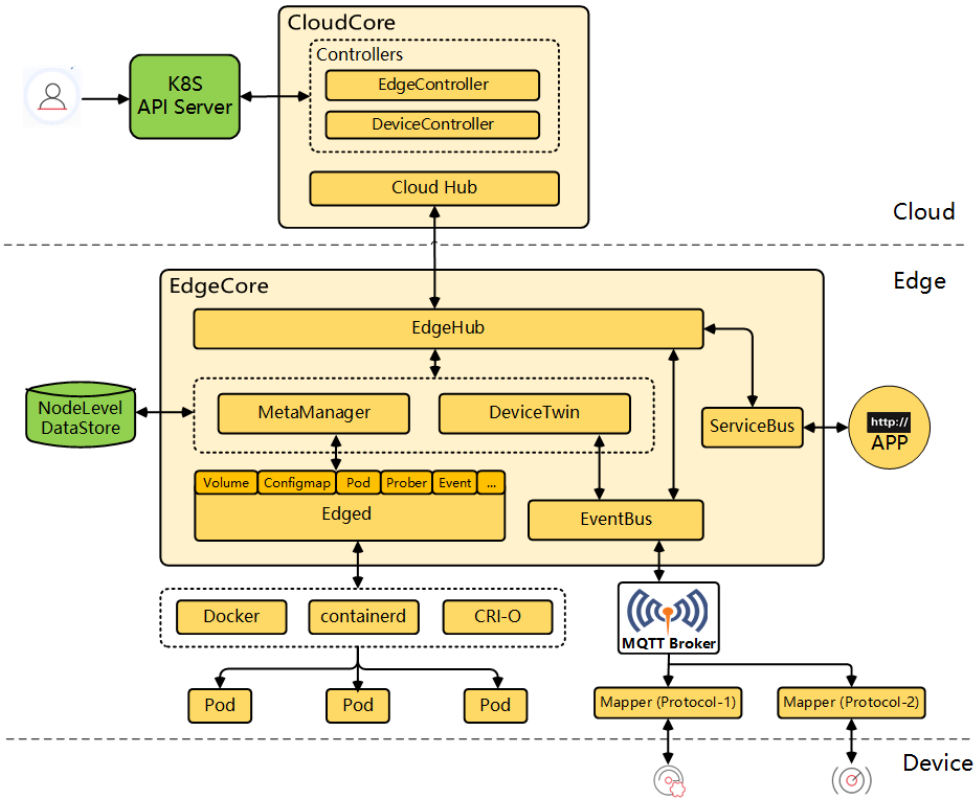

Figura 6. Arquitectura de KubeEdge.

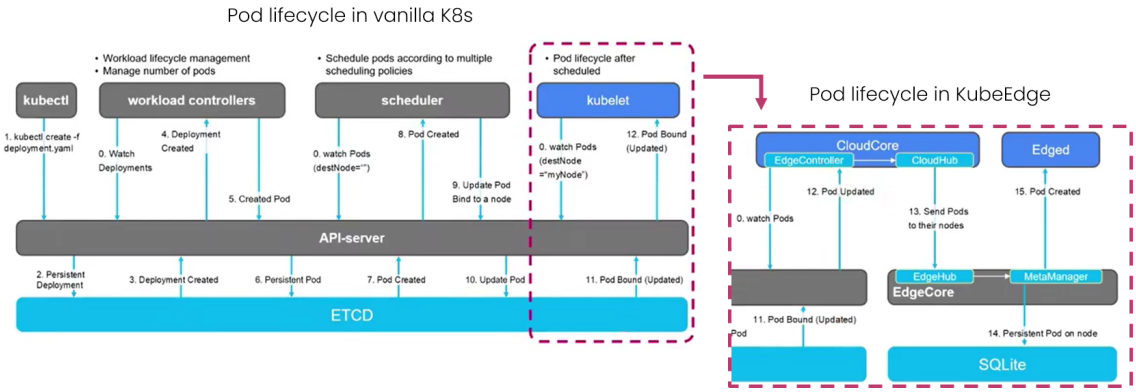

Figura 7. Ciclo de vida de los *pods* en *vanilla* Kubernetes y en KubeEdge.

Su distribución del procesamiento con control centralizado permite soportar nodos geográficamente dispersos. Existe una comunicación continua entre el *cloud* y el *edge*, pero soporta la autonomía del *edge*. En la Figura 8 se muestra el proceso de comunicación autónoma entre nodos *edge*, cuando el controlador del *cloud* no se encuentra disponible. En ella se muestran 2 procesos, el registro del servicio desde el nodo *cloud* en verde y el acceso al servicio de *edge* a *edge*, sin la intervención del nodo cloud, en rojo. La principal diferencia con un K8s tradicional radica en que la definición de los servicios se envía del *cloud* al *edge* después de definirse en el *cloud* por el usuario

y no cuando se invoca al servicio. De esta forma, cuando se invoca al servicio desde un nodo *edge*, este ya dispone de la información necesaria para encaminar la petición.

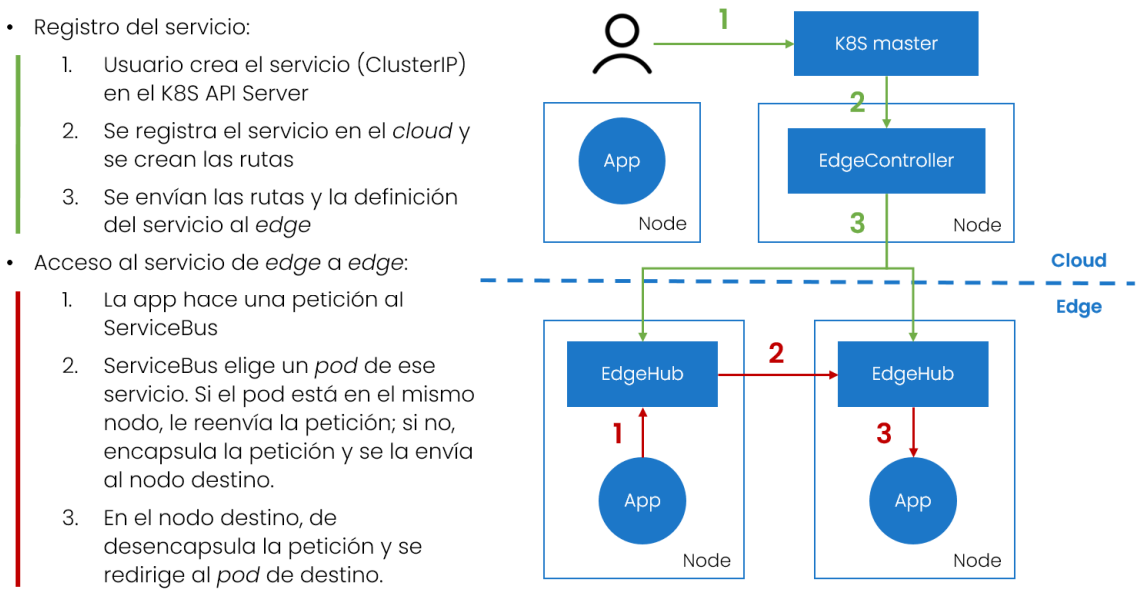

Figura 8. Proceso de comunicación autónoma entre nodos *edge*.

En resumen, KubeEdge es una distribución que resulta de gran interés para el *edge computing*, ya que tiene un alto nivel de madurez y continuidad de desarrollo y ofrece características muy útiles para entornos industriales y de IoT, que otras distribuciones no ofrecen, como el gemelo digital y la autonomía del *edge*. Sin embargo, requiere de un despliegue existente de K8s, ya sea mediante otra distribución o un *vanilla* K8s.

5.1.2. OpenYurt

OpenYurt es una distribución de código abierto de Kubernetes con origen en Alibaba, basada en su servicio propietario ACK@Edge que coordina el *cloud* con el *edge*. Actualmente, la ha adoptado el CNCF como un *sandbox project*. Sus principales ventajas son la compatibilidad completa con todos los objetos de la API de K8s, una fácil conversión de un *cluster vanilla* K8s a un *cluster* OpenYurt mediante su herramienta CLI *yurtctl*, el soporte de autonomía del *edge* y un enfoque agnóstico de despliegue en plataformas públicas *cloud*. Al igual que KubeEdge, permite una instalación ligera en dispositivos con recursos computacionales y almacenamiento limitados, pero también requiere de la existencia de un *cluster* K8s previo. En la Figura 9 se muestra la arquitectura en la que se basa [22] [23].

En el nodo controlador del *cloud* se despliegan 3 componentes:

- ***Yurt controller manager***: controlador que gestiona los nodos *edge*. Hace la función del servicio *kube-controller-manager* de un despliegue *vanilla* K8s. En el caso de pérdida de conexión con un nodo *edge*, no lo elimina de la base de datos, sino que lo habilita como nodo autónomo.
- ***Yurt app manager***: gestiona dos nuevos CRDs: **NodePool**, para la distribución geográfica por regiones, y **UnitedDeployment**, para la distribución de cargas de trabajo entre nodos de diferentes regiones. Ambos recursos se basan en la asignación de etiquetas a los nodos para el despliegue de aplicaciones. Los nodos de una misma región deberán tener una etiqueta que identifique de forma unívoca a la región a la que pertenecen. El término *pool* en OpenYurt hace referencia a una región en la que se distribuyen uno o varios nodos *edge*.
- ***Yurt tunnel server***: se conecta con el *Yurt tunnel agent* de cada nodo *edge* mediante un proxy inverso.

En cada uno de los nodos del *edge* se despliegan, además de un gestor de contenedores, 2 componentes específicos de esta distribución de K8s:

- **YurtHub**: servicio que hace de proxy para el tráfico saliente de los servicios de K8s (*kubelet*, *kube-proxy*, *plugins* CNI...) hacia el *cloud*. Cachea de forma local en el *edge* el estado de los recursos a los que acceden los servicios K8s, lo cual aporta cierta autonomía a los nodos *edge* cuando el *cloud* se desconecta.
- **Yurt tunnel agent**: se conecta con el *Yurt tunnel server* del nodo *cloud* mediante un proxy inverso.

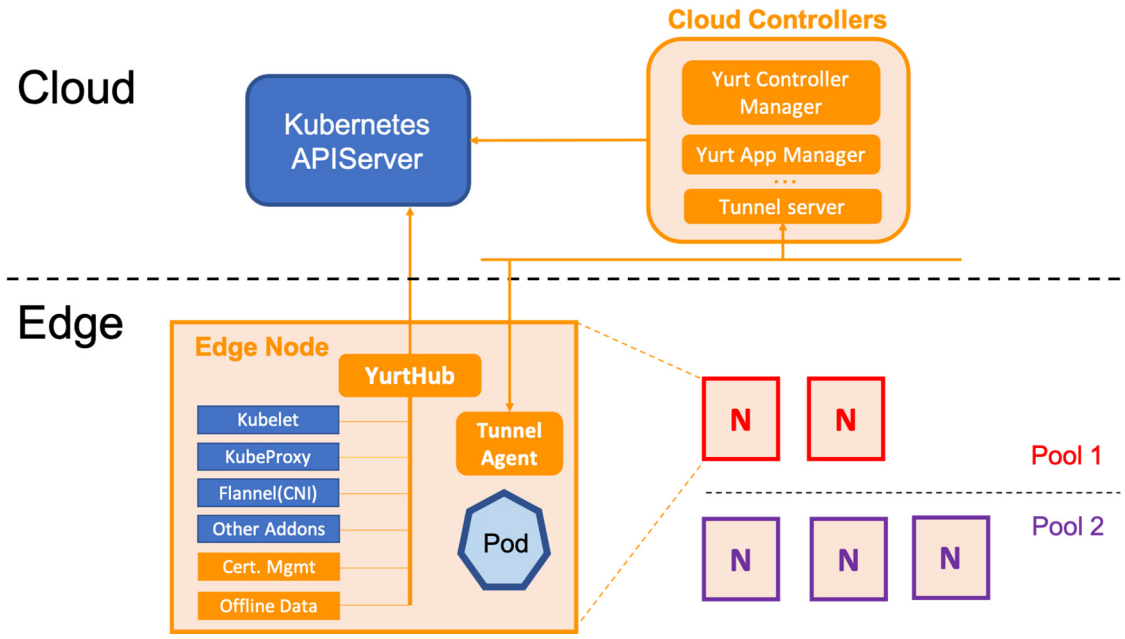

Figura 9. Arquitectura de OpenYurt.

Como ya se ha mencionado, OpenYurt soporta la distribución geográfica de nodos mediante los CRDs *NodePool* y *UnitedDeployment*. Por un lado, *NodePool* agrupa los nodos de una misma región mediante la asignación de una etiqueta que identifica a dicha región. Esta agrupación de nodos facilita la gestión y orquestación de los nodos y los servicios a desplegar en ellos, proporcionando un nivel adicional de abstracción. En la Figura 10 se muestra como ejemplo la distribución de 6 nodos en 2 regiones diferentes y su gestión unificada mediante una definición de *NodePool* por cada región [24].

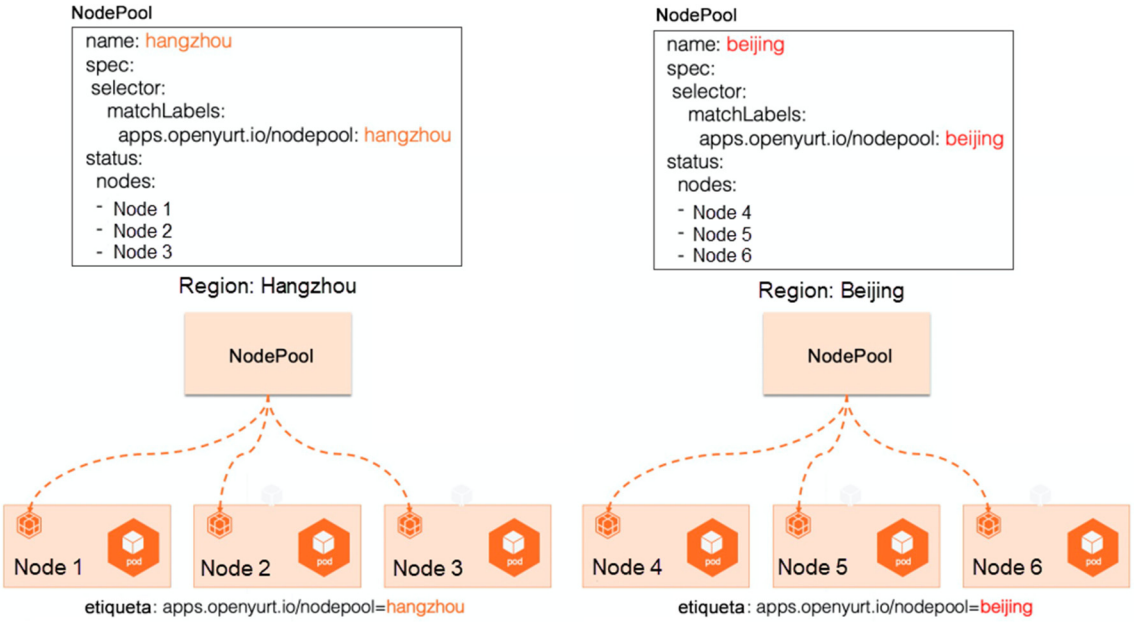

Figura 10. Ejemplo de uso de *NodePool* en OpenYurt.

Por otro lado, el objeto *UnitedDeployment* representa una abstracción de despliegue de cargas de trabajo de K8s, como *Deployments* o *StatefulSets*, en múltiples regiones. Esto permite desplegar una misma aplicación en varios nodos distribuidos, controlando este despliegue desde un único sitio. En la Figura 11 se muestra el mismo ejemplo de antes en el que se desean desplegar 2 réplicas de una misma aplicación en una región y 3 réplicas de la misma aplicación en la otra región. Este despliegue se controla mediante una única configuración de *UnitedDeployment*.

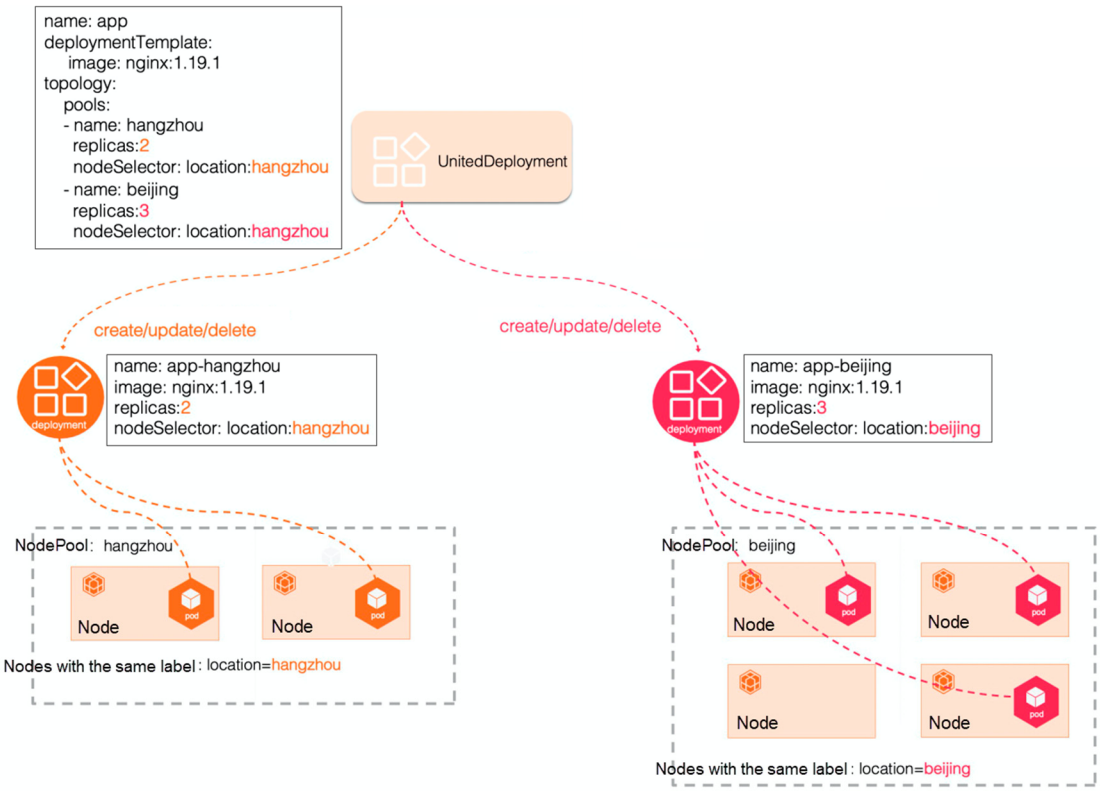

Figura 11. Ejemplo de uso de *UnitedDeployment* en OpenYurt.

El soporte de la distribución geográfica de nodos hace de OpenYurt una distribución a tener en cuenta en despliegues de *edge computing* a gran escala, en el que se han de gestionar múltiples sedes de una empresa de forma centralizada. Además, al estar certificada por la CNCF, es posible confiar en que tendrá un desarrollo de su código continuo con soporte de cara a futuro. Sin embargo, al igual que ocurre con KubeEdge, necesita de un despliegue base de K8s, lo cual implica tener que desplegar y gestionar múltiples distribuciones, ejecutadas de forma conjunta.

5.1.3. k3s

k3s es una distribución de código abierto de K8s con origen en Rancher, empresa de desarrollo de software para gestionar K8s a gran escala basada en California. Esta distribución está actualmente adoptada por el CNCF como *sandbox project*. Se diseñó para su uso en despliegues de producción con recursos limitados. Está optimizado para ARM (ARM64 y ARMv7), pero también soporta de forma nativa AMD64. Requiere de menos de 512 MB de RAM, por lo que se puede desplegar tanto en una Raspberry Pi como en un servidor *multicore*. Soporta múltiples distribuciones Linux, pero no es compatible con Windows.

A diferencia de las anteriores distribuciones de K8s descritas, esta sustituye la instalación de K8s, lo cual implica la necesidad de migrar el despliegue actual por completo. Se trata de una distribución muy ligera, que se empaqueta como un único

archivo binario de menos de 100 MB, con dependencias. Para aligerarla se eliminan ciertos componentes de K8s que son, en la mayoría de los despliegues, innecesarios.

Su arquitectura es similar a la de Kubernetes, con alguna pequeña diferencia. En k3s los nodos *master* reciben el nombre de *servers*, mientras que los nodos *worker* reciben el nombre de *agents*. En la Figura 6 se muestra la arquitectura en la que se basa.

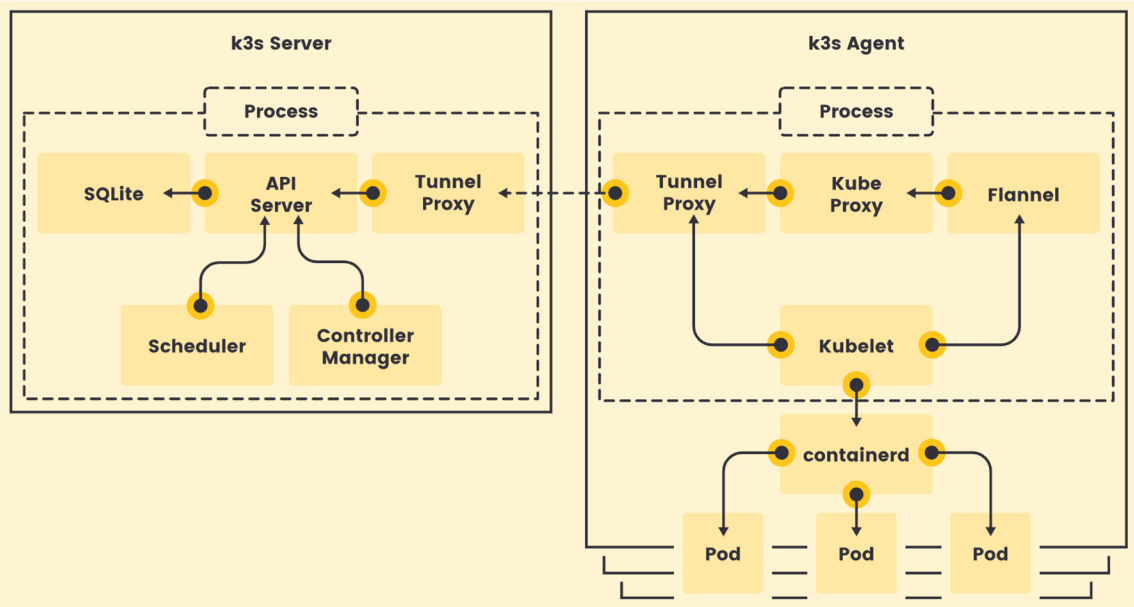

Figura 12. Arquitectura de k3s [25].

Las principales diferencias con Kubernetes son las siguientes [26]:

- k3s tiene un tamaño reducido de instalación y ficheros de configuración, lo cual agiliza el despliegue tanto del *cluster* como de las aplicaciones. Esto también permite aumentar la seguridad de los despliegues, al tener menos ficheros sobre los que atacar.
- K8s no es compatible con dispositivos *edge* o IoT, mientras que k3s soporta ARM64 y ARM7, ambas arquitecturas de CPU utilizadas por estos dispositivos.
- k3s no soporta de forma nativa la gestión de cargas de trabajo en múltiples entornos *cloud*. Para ello, requiere de herramientas adicionales como Rancher o Fleet.
- K8s utiliza el servicio *kube-proxy* para gestionar las conexiones entre contenedores y gestionar el enmascaramiento de direcciones IP, mientras que k3s solo lo utiliza para gestionar las conexiones entre nodos.

A pesar de que ofrece ciertas ventajas para entornos de *edge computing*, k3s no deja de ser una versión minimalista de un vanilla K8s, con facilidades de instalación. No ofrece ninguna característica adicional diferencial para casos de uso de *edge computing*.

5.1.4. MicroK8s

MicroK8s es una distribución de código abierto de Kubernetes con origen en Canonical, certificada por el CNCF como compatible con K8s. Al igual que k3s, se trata de una distribución ligera que minimiza la cantidad de añadidos que se incluirían de base en una instalación *vanilla* K8s. Es por eso que también sustituye por completo una instalación de K8s, obligando a migrar los *clusters* ya desplegados. Soporta arquitecturas de CPU tanto AMD como ARM, por lo que permite despliegues con Raspberry Pi. Se diseñó inicialmente para Ubuntu, pero incluye soporte para Windows y macOS. Su arquitectura es la misma que la de K8s, con el añadido de una mayor ligereza que permite despliegues más rápidos y en dispositivos con menos recursos [27].

Recientemente se están realizando cambios significativos sobre el diseño de esta distribución, por lo que aún no resulta una opción demasiado definida. Además, al igual que k3s, no aporta ningún valor adicional en cuanto a características de *edge computing*, más allá de su ligereza.

5.1.5. Kubermatic

Kubermatic es una empresa que ofrece dos distribuciones privativas de K8s: *Kubermatic Kubernetes Platform* y *KubeOne*. Ambas están certificadas por el CNCF como compatibles con K8s. La primera automatiza los despliegues de K8s para miles de *clusters*, con una gestión centralizada. Está específicamente diseñada para despliegues a gran escala [28]. *KubeOne*, en cambio, está optimizado para gestionar de forma automatizada el ciclo de vida de un único *cluster*, ya sea en el *cloud*, en el *edge* o en las premisas de un entorno IoT, como puede ser una fábrica [29].

Ambas son distribuciones para entornos muy específicos, la primera a gran escala y la segunda para despliegues pequeños. Su mayor desventaja radica en que no son distribuciones de código abierto. Sin embargo, Kubermatic es una de las empresas con mayores contribuciones al proyecto de Kubernetes, por lo que la compatibilidad está asegurada.

5.1.6. k0s

k0s (*Zero Friction Kubernetes*) es una distribución de código abierto de K8s con origen en Mirantis. No está certificada por el CNCF. Reduce la complejidad de instalación de K8s, pudiendo desplegar un *cluster* en unos pocos minutos sin necesidad de ninguna experiencia previa. Además, al igual que k3s, se distribuye como un único archivo binario con dependencias mínimas. Tuvo su lanzamiento inicial en noviembre de 2020, por lo que se trata de un proyecto que está todavía en sus primeras fases de desarrollo y no presenta actualmente características destacables frente a otras distribuciones [30].

5.1.7. SuperEdge

SuperEdge es una distribución de código abierto de K8s con origen en Tencent. No está certificada por la CNCF. Su principal característica es la distribución geográfica de nodos *edge*, gestionados de forma centralizada dentro de un mismo *cluster* desde el *cloud*. Ofrece, además, autonomía del *edge* para entornos de red inestables y monitorización constante de la salud de los nodos. Gracias al soporte de distribución multirregional de microservicios permite ofrecer servicios en bucle cerrado entre nodos *edge*, reduciendo la sobrecarga operativa y mejorando la disponibilidad del sistema [31].

Al igual que KubeEdge y OpenYurt, se despliega sobre un *cluster* de K8s existente. Es una distribución con una arquitectura similar a la de OpenYurt. Sin embargo, tiene menor madurez y soporte. En la Figura 13 se muestra la arquitectura en la que se basa.

En el nodo controlador del *cloud* se despliegan 3 componentes:

- **tunnel-cloud**: comunica el *edge* con el *cloud*. Mantiene una conexión persistente con los servicios del *tunnel-edge*. Soporta *proxies* TCP, HTTP y HTTPS.
- **application-grid-controller (ServiceGroup)**: controlador que gestiona dos nuevos CRDs: *DeploymentGrids* y *ServiceGrids*. Su funcionamiento es similar al de los objetos *NodePool* y *UnitedDeployment* de OpenYurt. También gestiona las aplicaciones y el tráfico de los nodos *edge*.
- **edge-admission**: gestiona los datos del estado de salud proporcionados por los servicios *edge-health* distribuidos en los nodos *edge*. Esto permite monitorizar las pérdidas de conexiones con los nodos *edge*.

En cada uno de los nodos del *edge* se despliegan, además de un gestor de contenedores, 4 componentes específicos de esta distribución de K8s:

- **lite-apiserver**: una implementación ligera del servicio *kube-apiserver* que aporta autonomía a los nodos *edge*.
- **edge-health**: monitoriza el estado de salud de los nodos *edge* de la misma región *edge* a la que pertenece el nodo.
- **tunnel-edge**: comunica el *edge* con el *cloud*. Mantiene una conexión persistente con el *tunnel-cloud* para obtener las peticiones API.
- **application-grid-wrapper**: proporciona un espacio interno de red independiente para los servicios dentro del mismo *ServiceGrid*. Se gestiona desde el *cloud*, a través del servicio *application-grid controller*.

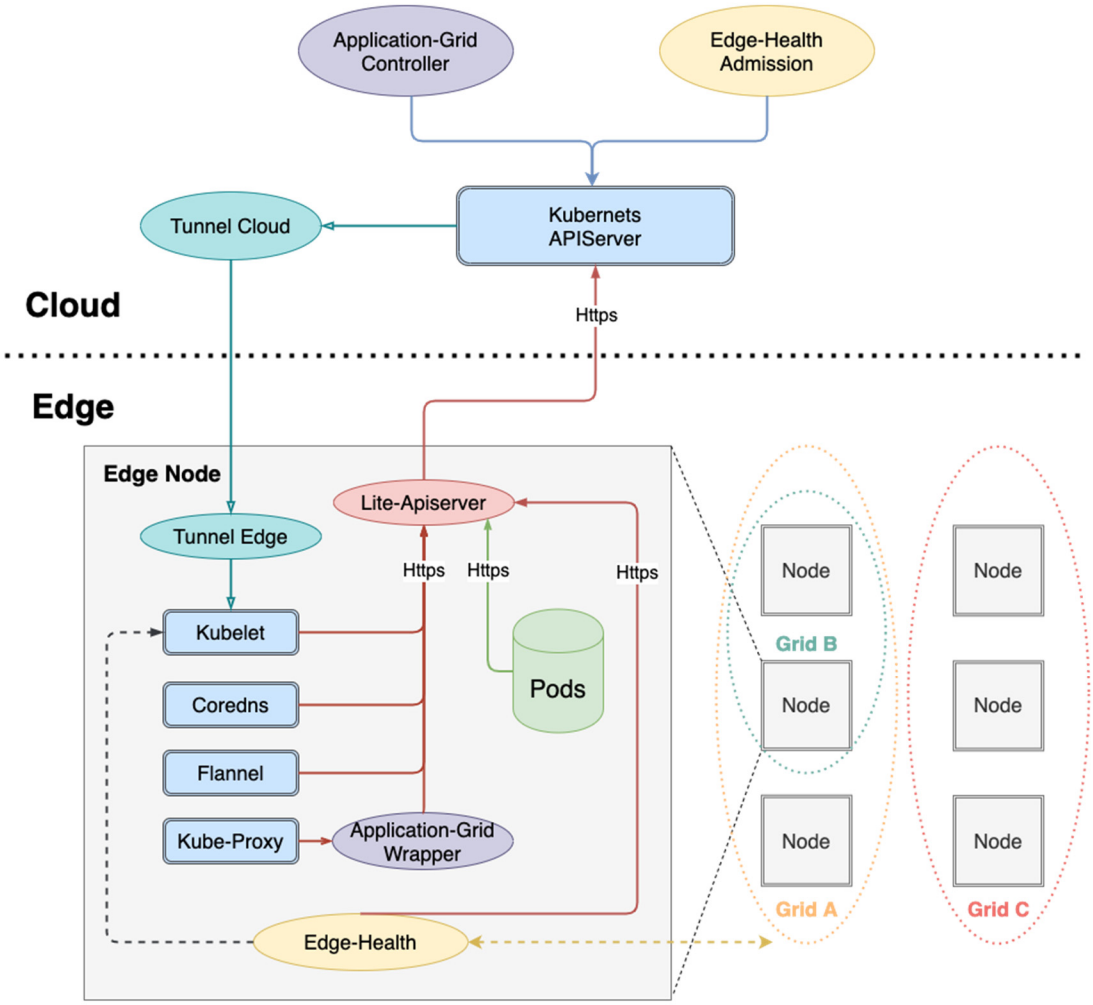

Figura 13. Arquitectura de SuperEdge.

6. Análisis de alternativas

En este apartado se analizarán las distribuciones de K8s de entre las descritas en el apartado para seleccionar la alternativa más adecuada y completa para un caso de uso de *edge computing*.

6.1. Descripción de las alternativas

Las opciones a considerar son las distribuciones de K8s para el *edge* descritas en el apartado anterior: KubeEdge, OpenYurt, k3s, MicroK8s, Kubermatic, k0s y SuperEdge. Algunas de estas propuestas permiten realizar un despliegue conjunto, por lo que sería posible seleccionar varias, si resultase ventajoso.

6.2. Descripción de los criterios de selección

Para la selección de la distribución de K8s a utilizar en el despliegue se han tenido en cuenta los siguientes criterios:

- **Nivel de madurez de la solución**, definido por la CNCF. Como ya se ha comentado en apartados anteriores, la CNCF entrega a los proyectos una etiqueta en base a su nivel de madurez. Hay ciertos proyectos que están enmarcados bajo dichas etiquetas, otros a los que simplemente se les certifica de cierta forma que son compatibles con el ecosistema Kubernetes y otros que no disponen de ningún tipo de etiqueta ni certificación. Como punto inicial, se han descartado como alternativas las soluciones presentadas en el apartado de estado del arte que no disponen de un nivel de madurez *graduated project*, *incubating project* o *sandbox project*. Aquellas soluciones descartadas en este punto no se analizarán bajo los siguientes criterios. Para el análisis de las alternativas restantes, a este criterio se le ha dado un peso del 20%. Cuanto más madura sea una distribución, mejor. Se puntuará con un 10 a aquellas que sean *graduated project*, con un 6 aquellas que sean *incubating project* y con un 3 aquellas que sean *sandbox project*.
- **Utilización de recursos**. Cuantos menos recursos consuma, mejor se adaptará a los dispositivos del *edge*, que tienen capacidades limitadas. Dentro de este criterio se tendrá en cuenta si se trata de una instalación mínima, así como el consumo de recursos una vez instalado y desplegado. A este criterio se le ha dado un peso del 20%.
- **Rendimiento**. El rendimiento es otro de los aspectos clave que sirve como referencia para saber cómo afecta la ejecución de una aplicación al sistema. Para valorar el rendimiento de las distribuciones a analizar, se realizarán varias pruebas siguiendo una metodología, descrita en el siguiente subapartado. A este criterio se la ha dado un peso del 30%.

— **Capacidades adicionales** ventajosas para casos de uso en el *edge*. Se priorizarán aquellas distribuciones que dispongan características especialmente dirigidas a despliegues en el *edge*, como el soporte de arquitecturas ARM, la gestión de dispositivos o la autonomía de los nodos *edge*. A este criterio se le ha dado un peso del 30%.

6.3. Metodología seguida para medir el rendimiento y la utilización de recursos

El objetivo de las pruebas es medir de forma sistemática el rendimiento de la CPU, de los accesos a memoria y de las lecturas y escrituras del disco, así como la utilización de CPU y memoria que hace cada distribución.

La metodología de pruebas se ha dividido en 3 partes: la captura de datos, el procesamiento de datos y la visualización de datos para la generación del informe de resultados.

6.3.1. Captura de datos

A continuación se describen de forma general las herramientas utilizadas, los indicadores clave de rendimiento (KPI, *Key Performance Indicator*) a tener en cuenta en cada prueba y los detalles específicos de cada prueba a realizar.

Para poder dar como válidos los resultados, se han ejecutado las pruebas 30 veces, midiendo cada indicador en cada una de las iteraciones y obteniendo el valor medio como resultado final. De esta forma, eliminamos cualquier valor extremo que pueda afectar a la comparativa y podemos comprobar que la desviación estándar de los datos del conjunto es baja. Para asegurar esto último, se comprueba que la media del conjunto de datos de todas las iteraciones entra dentro de un rango o intervalo de confianza. Si la desviación estándar tiene un valor bajo y la media entra dentro de ese intervalo de confianza, podemos asegurar que los resultados de las pruebas son consistentes [32].

6.3.1.1. Herramientas de medida

En base al tipo de medida a realizar, se han utilizado herramientas diferentes:

— Para las medidas de utilización de recursos de CPU y memoria se utilizará el **comando ps** de Linux.
— Para medir el rendimiento de CPU y memoria se utilizará la herramienta **Sysbench**.
— Para medir el rendimiento de disco se hará uso de la herramienta **FIO**.

COMANDO PS

El comando `ps` de Linux se utiliza para mostrar información sobre los procesos activos del sistema. Para mostrar los procesos de todos los usuarios del sistema, el comando se utiliza de la siguiente forma: "`ps aux`". A la salida se muestra algo similar a lo de la Figura 14. En dicha salida, cada columna representa lo siguiente [33]:

- USER: usuario al que pertenece el proceso.
- PID: identificador del proceso.
- %CPU: porcentaje de utilización de la CPU por parte de este proceso. Este porcentaje se calcula como el tiempo de CPU utilizado sobre el tiempo que el proceso lleva ejecutándose.
- %MEM: porcentaje de utilización de la memoria RAM por parte de este proceso. Este porcentaje se calcula como el ratio entre el tamaño de espacio de RAM asignado al proceso, también llamado RSS (*Resident Set Size*), entre el tamaño total de la RAM.
- VSZ: utilización de memoria virtual del proceso, en KiB.
- RSS: tamaño de espacio de RAM total asignado al proceso, en KiB.
- TTY: terminal que controla el comando que ha activado el proceso.
- STAT: estado del proceso.
- START: hora o fecha de inicio del proceso.
- TIME: tiempo acumulativo de uso de CPU por parte del proceso.
- COMMAND: comando completo que ha iniciado el proceso.

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
cristina    1434  0.2  0.4  18732  9988 ?        Ss   09:29   0:00 /lib/systemd/systemd --user
cristina    1435  0.0  0.1 103464  3440 ?        S    09:29   0:00 (sd-pam)
cristina    1440  0.0  0.2   7072  5044 tty1     S    09:29   0:00 -bash
cristina    4219  0.0  0.1   7648  3228 tty1     R+   09:33   0:00 ps aux
```

Figura 14. Salida de ejemplo del comando `ps aux`.

SYSBENCH

Sysbench es una herramienta que permite medir el rendimiento de un sistema de forma directa y sencilla. Dispone de una colección de pruebas prediseñadas para medir el rendimiento de CPU, memoria, disco, bases de datos, etc. Además, dado que está presente en los repositorios de la mayoría de distribuciones Linux, su instalación es muy sencilla. Se basa en la creación de uno o varios hilos que ejecutan de forma paralela unas peticiones. El trabajo que ejecuten estos hilos dependerá del tipo de prueba especificada. Para limitar la ejecución se puede especificar tanto un número total de eventos (por defecto, ilimitados) como el tiempo límite de ejecución de la prueba (por defecto, 10 segundos). En el caso de especificar ambos, el primer límite que se alcance marcará el fin de la prueba [34].

FIO

FIO (*Flexible I/O tester*) es una herramienta diseñada para medidas de rendimiento de disco, a base de simulaciones de carga de trabajo. Para ello, genera una serie de hilos o procesos que realizan una acción determinada de entrada y salida, según las opciones especificadas por el usuario en la línea de comandos.

Se le pueden especificar varios parámetros, como el tipo de simulación (lectura, escritura, secuencial, aleatoria, mixta, etc.), el tamaño de bloque, la ruta del fichero de simulación, etc. De acuerdo con la métrica que se desea medir (IOPS, *throughput*, latencia, etc.) será mejor una configuración u otra.

6.3.1.2. Indicadores clave de rendimiento (KPI)

Los indicadores clave de rendimiento o KPI son métricas que permiten medir los resultados de pruebas y determinar si son o no satisfactorias, en base a los parámetros seleccionados como más relevantes. Los KPI en los que se basarán las pruebas a realizar se resumen, para que sirva de referencia, en la Tabla 1. En los siguientes subapartados se describirá cómo se medirá cada uno de los indicadores.

Tabla 1. Resumen de los KPI a tener en cuenta en las pruebas.

Tipo de prueba	Nombre de KPI	Herramienta de medida
CPU	Porcentaje de utilización	Comando `ps`
	Eventos por segundo	Sysbench
Memoria	Porcentaje de utilización	Comando `ps`
	Throughput de escritura	Sysbench
	Operaciones de escritura por segundo	
	Throughput de lectura	
	Operaciones de lectura por segundo	
Disco	*Throughput* de escritura	FIO
	Operaciones de escritura	
	Latencia de escritura	
	Throughput de lectura	
	Operaciones de lectura	
	Latencia de lectura	

CPU

Para obtener el porcentaje de utilización de la CPU mediante el comando `ps`, se buscará el proceso o los múltiples procesos ejecutados por la distribución y se anotará el valor de la columna %CPU. Esta medida, en vez de realizarla de forma iterativa sin variar el entorno, se hará incrementando el número de *pods* desplegados con la distribución de Kubernetes a analizar. De esta forma, se podrá evaluar cómo afecta la cantidad de *pods* desplegados a la utilización de la CPU.

En las pruebas de rendimiento de CPU de Sysbench cada petición consiste en el cálculo de números primos hasta un límite que puede ser especificado y que, por defecto, es 10000. Para ello, se realizan consecuentes verificaciones dividiendo cada número por todos los números entre el 2 y su raíz cuadrada. Si alguno de los números tiene un resto de 0, se calcula el siguiente número. Todos los cálculos se realizan utilizando números enteros de 64 bits. Cada hilo ejecuta las peticiones de forma concurrente hasta que se alcanza el número total de peticiones o el tiempo límite de ejecución.

De entre todos los indicadores medidos por Sysbench en la prueba, el más relevante para la evaluación del rendimiento de la CPU es el número de eventos por segundo. Esto es debido a que se ha decidido mantener el límite por defecto de la herramienta, que, como ya se ha dicho, es el tiempo de ejecución. Si el límite se estableciese en el número total de eventos, el indicador a utilizar debería ser el tiempo de ejecución de la prueba. Cuanto mayor sea el número de eventos por segundo que es capaz de realizar, mejor.

MEMORIA

Para obtener el porcentaje de utilización de la memoria RAM mediante el comando `ps` se buscará el proceso o los múltiples procesos ejecutados por la distribución y se anotará el valor de la columna %MEM. Al igual que para la prueba de utilización de CPU, en vez de realizar la medida de forma iterativa sin variar el entorno, esta se hará incrementando el número de *pods* desplegados con la distribución de Kubernetes a analizar. De esta forma, se podrá evaluar cómo afecta la cantidad de *pods* desplegados a la utilización de la memoria.

En las pruebas de rendimiento de memoria, Sysbench mide el rendimiento del sistema frente a lecturas y escrituras secuenciales o aleatorias en memoria. Se pueden especificar parámetros como el tamaño de bloque a utilizar, el tipo de acceso que tendrán los hilos (global o local), el tipo de operación a ejecutar (lectura o escritura) o el modo de acceso a memoria (secuencial o aleatorio).

Los indicadores relevantes en este caso son el *throughput* y el número de operaciones por segundo, tanto para la lectura como para la escritura. Cuantos más altos sean estos valores, mayor rendimiento tendrá el sistema. Por simplificar y reducir el número de pruebas, se han hecho medidas únicamente para lecturas y escrituras aleatorias y no secuenciales.

Disco

Los indicadores que suelen tomar mayor relevancia en las pruebas de rendimiento de operaciones de escritura y lectura de disco son los siguientes:

- ***Throughput***: mide la cantidad de información que un dispositivo o sistema puede transferir en un periodo de tiempo. Generalmente se mide en Mbps o Gbps. Cuanto más alto sea este indicador, mejor.
- **Operaciones por segundo (IOPS, *I/O Operations Per Second*)**: mide la capacidad del sistema de almacenamiento de procesar peticiones de entrada y salida. Cuantas más operaciones por segundo procese, mayor rendimiento tendrá.
- **Latencia**: mide el tiempo que se tarda en completar una única operación de entrada y salida, desde el punto de vista de la aplicación. Se suele medir en milisegundos y cuanto menor sea su valor, mejor.

Generalmente, se suelen medir la latencia y las IOPS utilizando tamaños de bloques pequeños (4 KB) y una profundidad de cola pequeña. El *throughput*, en cambio, se suele medir con tamaños de bloque más grandes y mayor profundidad de colas. En estas pruebas de disco es importante tener en cuenta que el tamaño total debe ser superior a la RAM disponible en cada nodo.

6.3.2. Procesamiento de datos

Las pruebas con el comando `ps` se han ejecutado de forma manual, mientras que las que hacen uso de las herramientas Sysbench y FIO se han ejecutado de forma automatizada con un *script*, que se incluye como referencia en el Anexo II.

El *script* mencionado está escrito en el lenguaje de programación bash y se encarga de ejecutar los comandos correspondientes de las herramientas Sysbench y FIO para cada tipo de prueba. Para medir los diferentes KPIs es necesario utilizar diferentes configuraciones de dichos comandos. Tras la ejecución de dichos comandos, se filtra la salida obtenida para obtener el dato específico de la métrica. Por último, los datos obtenidos se escriben en un fichero CSV para su posterior visualización.

Como ejemplo, se describirá el proceso que se realiza dentro del *script* para medir el número de eventos por segundo de las pruebas de rendimiento de CPU. En este caso, el comando de ejecución de la prueba es el siguiente:

```
sysbench cpu run
```

Si ejecutásemos ese comando directamente desde una terminal, la salida que se obtendría tendría la siguiente forma:

```
sysbench 1.0.18 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 1
Initializing random number generator from current time


Prime numbers limit: 10000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:  1290.60

General statistics:
    total time:                          10.0001s
    total number of events:              12908

Latency (ms):
         min:                                    0.67
         avg:                                    0.77
         max:                                    3.91
         95th percentile:                        0.92
         sum:                                 9997.61

Threads fairness:
    events (avg/stddev):           12908.0000/0.00
    execution time (avg/stddev):   9.9976/0.00
```

Sin embargo, no nos interesa toda la salida, sino únicamente el valor del número de eventos por segundo que se han dado en la prueba. Para ello, el *script* filtra dicha salida, buscando la fila que contiene la frase 'events per second' mediante el comando `grep` y obteniendo la 4ª columna mediante el comando `awk`:

```
CPU_OUTPUT=$(sysbench cpu run)
EVENTS_PER_SECOND=$(echo "${CPU_OUTPUT}" | grep "events per second" | awk '{print $4}')
```

Una vez hecho esto, almacena el dato en el fichero CSV correspondiente:

```
CPU_FILE="$BASE_DIR/cpu_test_results-${CURRENT_TIME}.csv"
echo "${EVENTS_PER_SECOND}" >> "${CPU_FILE}"
```

6.3.3. Visualización de datos

Como ya se ha mencionado, los resultados de las pruebas se almacenan en ficheros CSV, uno por cada tipo de prueba (rendimiento de CPU, rendimiento de escritura en memoria, rendimiento de lectura de memoria y rendimiento de disco). El almacenar estos datos en formato CSV facilita su posterior procesado y visualización gráfica mediante herramientas de hojas de cálculo, como Excel.

Para este proyecto, se han producido los gráficos para el informe de resultados de forma manual, subiendo los CSV a la herramienta Flourish [35], disponible en línea, y modificando el formato de visualización a mano.

De cara a una ampliación futura de la metodología, podría plantearse realizar un *script* de visualización en Python que utilice librerías específicas de visualización, como `csv`, `matplotlib`, `plotly`, etc. Para la gestión de esos datos también se podría utilizar la librería `numpy`, siendo de especial utilidad para operar sobre grandes cantidades de datos. Esta librería gestiona los *arrays* de una forma más rápida y compacta que las listas nativas de Python, consumiendo menos memoria.

6.4. Selección de las alternativas

6.4.1. Nivel de madurez de la solución

Como ya se ha comentado, se descartan aquellas distribuciones que no están certificadas con una de las 3 etiquetas de madurez del CNCF. Por lo tanto, quedan fuera del análisis las distribuciones MicroK8s, Kubermatic, k0s y SuperEdge. Las distribuciones que quedan dentro del análisis son las siguientes:

- KubeEdge, clasificado como *incubating project*.
- OpenYurt, clasificado como *sandbox project*.
- k3s, clasificado como *sandbox project*.

De esas 3 distribuciones, la que tiene mayor nivel de madurez es KubeEdge. Hay que tener también en cuenta que tanto KubeEdge como OpenYurt requieren de una instalación base de K8s, que podría ser el propio K8s o la versión minimizada que ofrece k3s. La versión vanilla K8s está clasificado como un proyecto *graduated*, con el máximo nivel de madurez.

6.4.2. Utilización de recursos y rendimiento

Partiendo de las alternativas no descartadas en el subapartado anterior, en este subapartado se describirá de forma básica y a un alto nivel las arquitecturas de las pruebas de rendimiento y de medición de utilización de recursos ejecutadas sobre ellas. El diseño de más bajo nivel de estas pruebas se detalla en el Anexo II. Después, se mostrarán los resultados obtenidos en las pruebas.

6.4.2.1. Arquitectura de las pruebas

Se estudiará el rendimiento de 5 arquitecturas, cada una con una combinación diferente de las distribuciones K8s a analizar. De esta forma, se pretenden observar las ventajas que supone cada combinación frente a las otras. Dichas arquitecturas son las mostradas en la Figura 15 y se describen a continuación:

1. ***Standalone k3s***: se utilizará un nodo *server* k3s en el *cloud* y un nodo *agent* k3s en el *edge*. El análisis de esta arquitectura tiene el objetivo de extraer posibles ventajas de esta distribución más ligera frente a la de Kubernetes tradicional.
2. ***KubeEdge sobre Kubernetes***: por un lado, en el *cloud* se realizará un despliegue tradicional de un nodo *master* Kubernetes, sobre el cual se instalará un *CloudCore* de KubeEdge; por otro lado, en el *edge* se utilizará un nodo en el que se instalará únicamente la herramienta CLI de Kubernetes y un *EdgeCore* de KubeEdge.
3. ***KubeEdge sobre k3s***: similar al caso anterior, en el *cloud* se realizará un despliegue de un nodo *server* k3s, sobre el cual se instalará un *CloudCore* de KubeEdge; mientras que en el *edge* se utilizará un nodo en el que se instalará únicamente la herramienta CLI de Kubernetes y un *EdgeCore* de KubeEdge.
4. ***OpenYurt sobre Kubernetes***: al igual que con KubeEdge, en el *cloud* se desplegará Kubernetes, sobre el cual se instalará el controlador de OpenYurt, y en los nodos edge se instalará únicamente la herramienta CLI de Kubernetes y el *YurtHub* de OpenYurt.
5. ***OpenYurt sobre k3s***: igual que el caso anterior, pero desplegando el servidor de k3s en el nodo *cloud*.

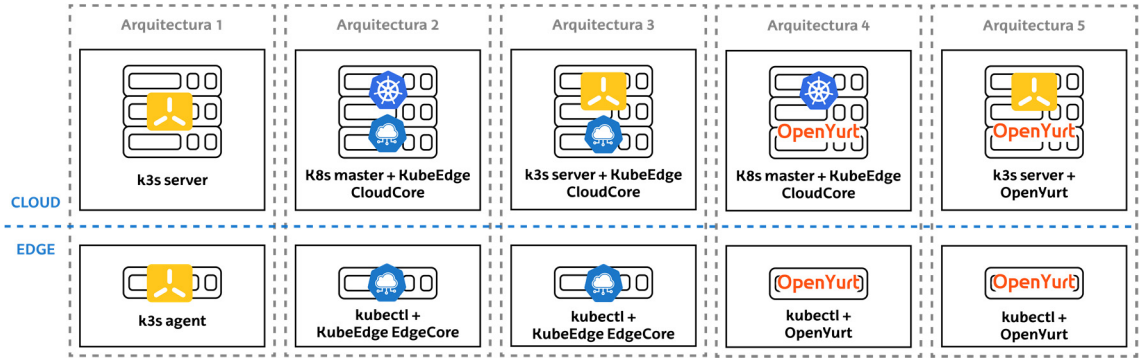

Figura 15. Esquema de las 5 arquitecturas a desplegar para las pruebas.

Las pruebas se han realizado sobre 3 nodos Ubuntu 20.04, uno de ellos como nodo controlador, simulando la parte *cloud*, y los otros 2 como nodos *edge*. Se han utilizado nodos de diferentes características o especificaciones físicas, de tal forma que se pueda estudiar el impacto que tienen estas sobre cada caso de uso. Sus principales características son las siguientes:

— El nodo *cloud* está desplegado en una máquina virtual alojada en el nodo OpenStack del laboratorio, a la que se le han asignado 8 cores, 16 GB de RAM y 200 GB de almacenamiento. Estas características son similares a las que podría tener un nodo *cloud* en un entorno de producción.

— Uno de los nodos *edge* se ha desplegado sobre un servidor Supermicro con 12 cores, 64 GB de RAM y 477 GB de almacenamiento. Este servidor sirve como réplica o referencia a lo que se podría desplegar en un entorno real en el *edge*, cuando no existen excesivas limitaciones económicas, de movilidad o de espacio.
— El otro nodo *edge* se ha desplegado sobre una Raspberry Pi 3 con 4 cores, 1 GB de RAM y 8 GB de almacenamiento de tarjeta microSD. Dado que es un nodo con unos recursos muy limitados, su óptima utilización es de gran importancia.

Las pruebas de utilización de recursos y rendimiento se han ejecutado únicamente sobre los nodos *edge*, ya que son los que se ven más limitados en un entorno real.

6.4.2.2. Resultados de utilización de recursos

Primero se analizará la utilización de recursos del nodo *edge* Supermicro, para describir a continuación la utilización de recursos analizada en el nodo *edge* Raspberry Pi 3.

En la Figura 16 se muestra el porcentaje de uso de la CPU en el nodo *edge* Supermicro para cada distribución a analizar. Como se puede observar, en todas ellas existe muy poca variación del porcentaje desde no tener ninguna réplica desplegada hasta las 100 réplicas desplegadas. La arquitectura con peor rendimiento en este caso es la de k3s *standalone*, que consume un 1% más con 100 réplicas que con ninguna. Sin embargo, la variación no es especialmente relevante.

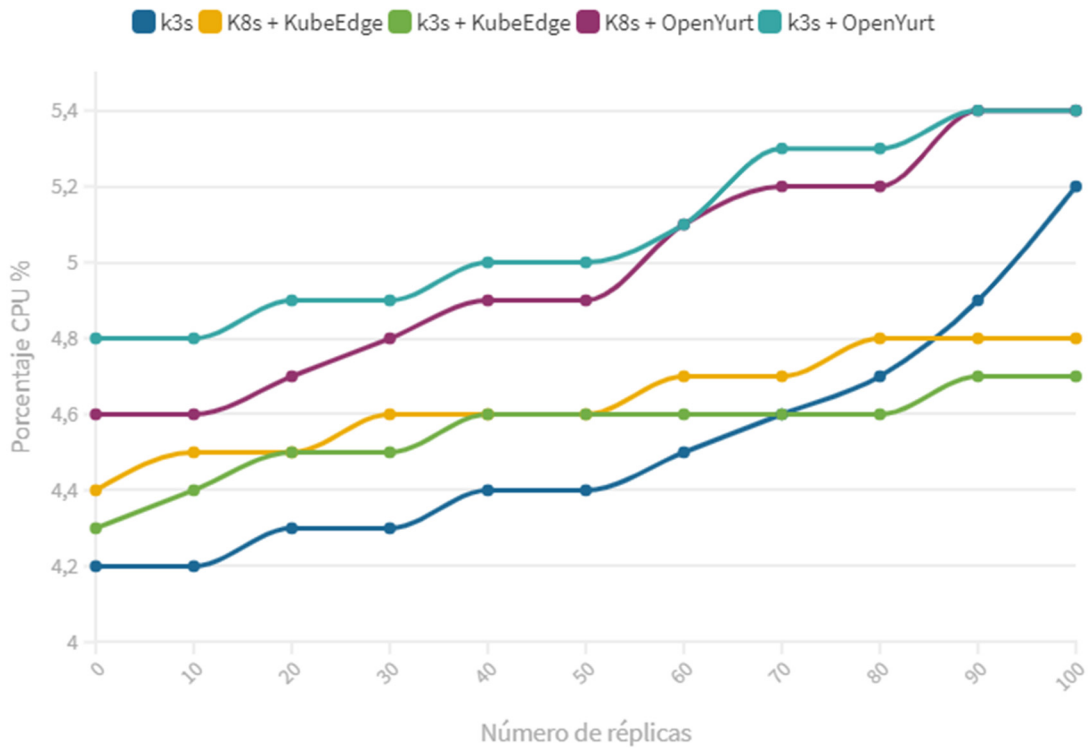

Figura 16. Porcentaje de uso de la CPU en el nodo Supermicro.

En la Figura 17 se muestra el porcentaje de uso de la memoria del Supermicro. Al igual que para la CPU, no existe una variación relevante, aunque también destaca ligeramente a peor la arquitectura de k3s *standalone*.

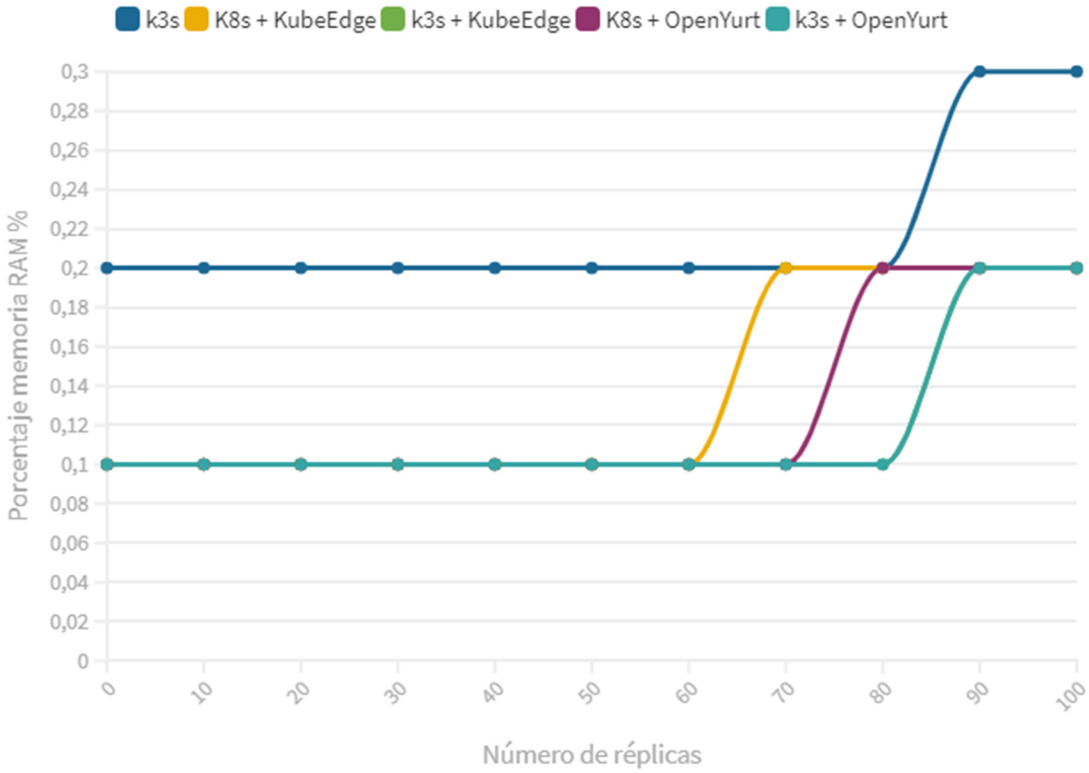

Figura 17. Porcentaje de uso de la memoria RAM en el nodo Supermicro.

Para el nodo *edge* Raspberry Pi 3, se muestra en la Figura 18 el porcentaje de uso de CPU de cada arquitectura analizada. El uso de la CPU aumenta de forma lineal en todos los casos, con un porcentaje de uso menor para las arquitecturas con KubeEdge. Cabe destacar que, a diferencia del nodo Supermicro, en el que se habían desplegado hasta 100 réplicas, el máximo número de réplicas que la Raspberry Pi 3 ha permitido han sido 30. Una vez superado un porcentaje de uso de la CPU del 35%, por parte del proceso de la distribución de Kubernetes que se ejecuta en el nodo, el dispositivo entraba en un estado indefinido y dejaba de funcionar.

De forma análoga, la Figura 19 muestra cómo no existe prácticamente variación en el uso de la memoria RAM para ninguna de las arquitecturas, a medida que se aumenta el número de réplicas.

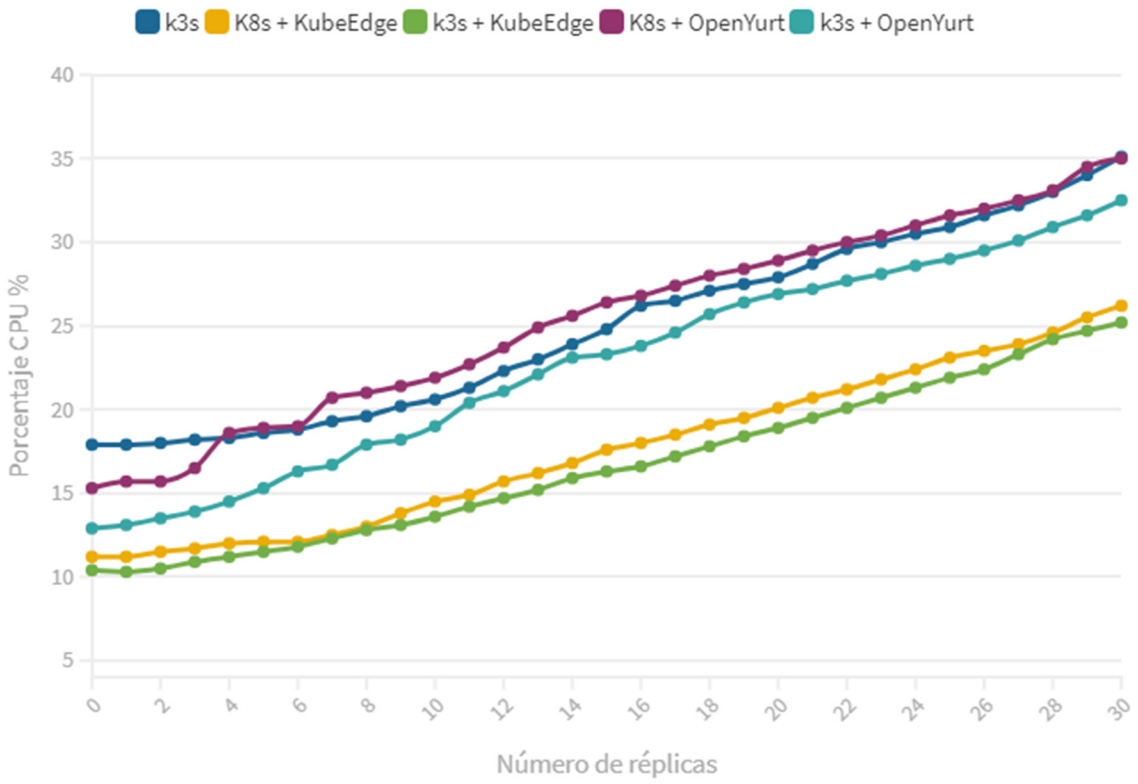

Figura 18. Porcentaje de uso de la CPU en el nodo Raspberry Pi 3.

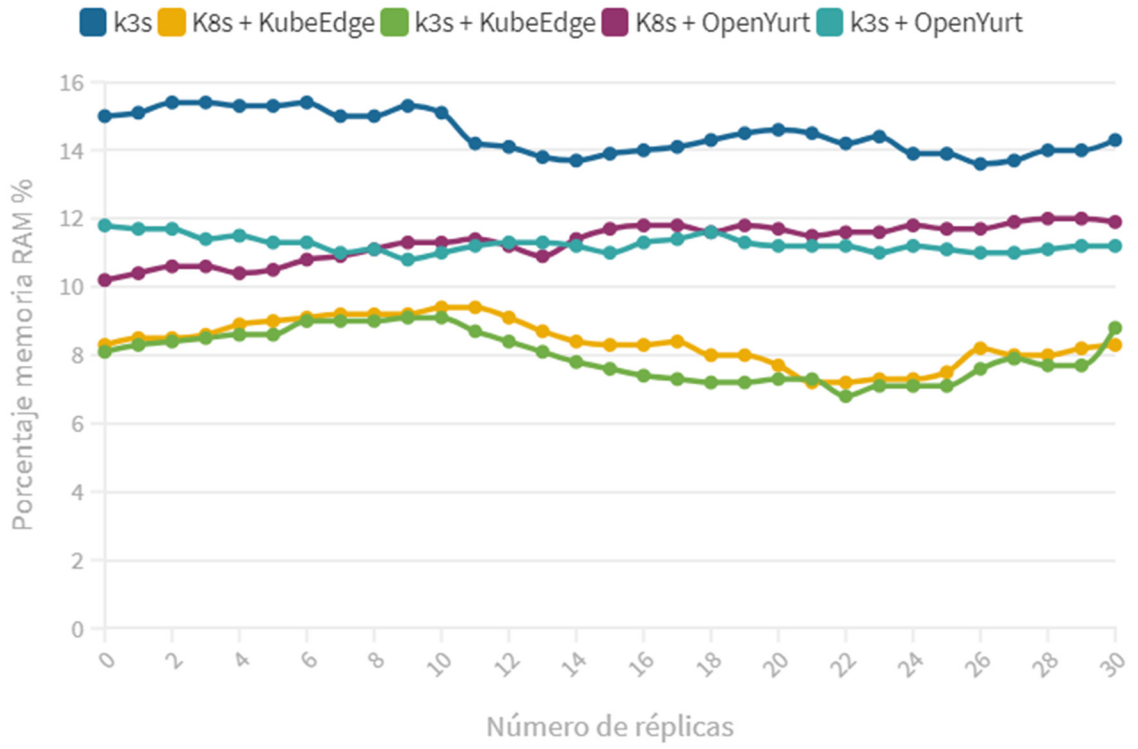

Figura 19. Porcentaje de uso de la memoria RAM en el nodo Raspberry Pi 3.

6.4.2.3. Resultados de las pruebas de rendimiento

Al igual que para los resultados de la utilización de recursos, primero se mostrarán los resultados de rendimiento del nodo *edge* Supermicro, para describir a continuación los del nodo *edge* Raspberry Pi 3.

En la Figura 18 se muestran los resultados de los KPIs de rendimiento para la CPU y la memoria RAM del nodo Supermicro. Se puede observar que no existe demasiada variación entre distribuciones. Esta poca variación general se debe a que el nodo Supermicro no está demasiado limitado ni el CPU, ni en memoria, por lo que ofrece bastante margen a las aplicaciones que corren en él.

En lo que respecta a rendimiento en disco, en la Figura 21 se puede observar que el número de IOPS de escritura empeora considerablemente cuando existe un despliegue de Kubernetes, hasta un 33% en el caso del k3s *standalone*. Las arquitecturas que presentan menos variación y, por lo tanto, mejor rendimiento, son la de K8s + KubeEdge y k3s + KubeEdge.

En la lectura la diferencia es menos evidente, pero siguen obteniendo mejor rendimiento aquellas arquitecturas con KubeEdge, independientemente de la distribución de Kubernetes que se ejecuta en el nodo *cloud*.

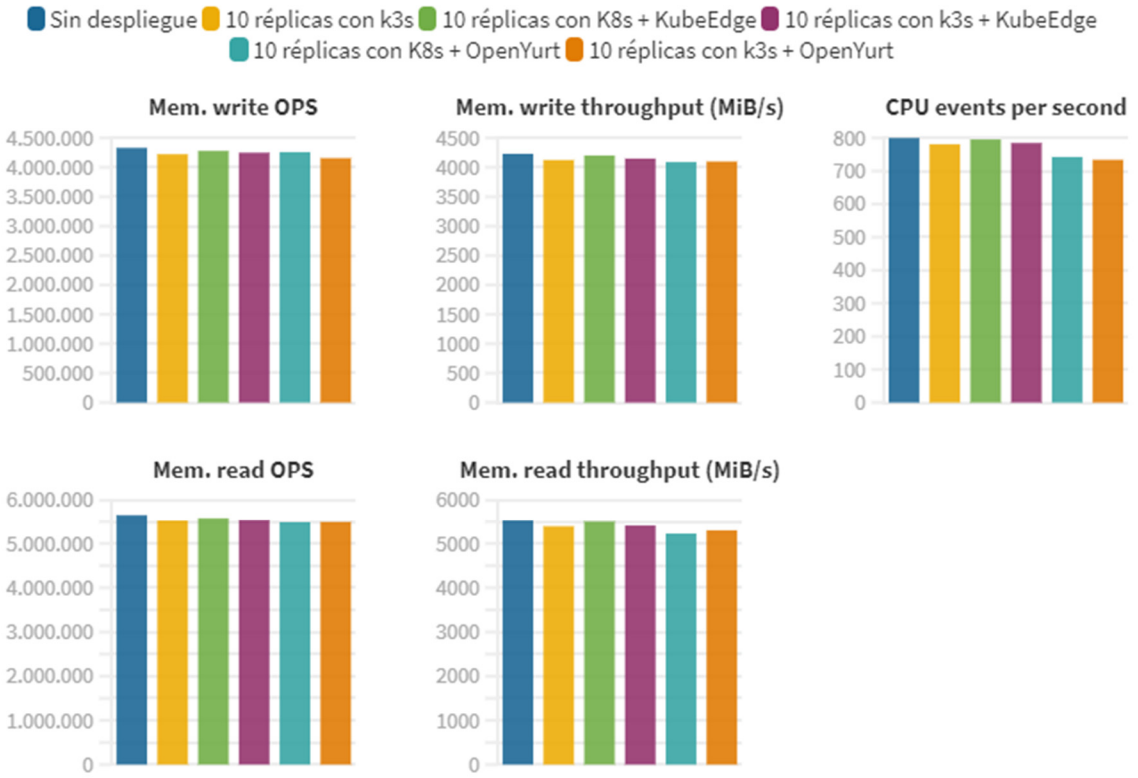

Figura 20. Rendimiento de memoria y CPU con Sysbench para el nodo Supermicro.

Figura 21. Rendimiento de disco con FIO para el nodo Supermicro.

En cuanto al rendimiento de CPU y memoria de la Raspberry Pi 3, no se han observado diferencias relevantes entre arquitecturas, aunque sí se puede observar un decremento del rendimiento de todas ellas respecto a un nodo sin ningún despliegue. Esto se muestra en la Figura 22.

Por último, en la Figura 23 se muestran los resultados obtenidos para el rendimiento de disco con la herramienta FIO en la Raspberry Pi 3. A diferencia que en el caso del Supermicro, el despliegue de cualquiera de las arquitecturas empeora significativamente la lectura de disco. También empeora la escritura de disco, pero con menos diferencias. De nuevo, las arquitecturas que destacan como mejores en el rendimiento de disco son aquellas que tienen KubeEdge en el nodo *edge*, aunque el k3s *standalone* obtiene mejores resultados comparativos en la Raspberry Pi 3 que en el Supermicro.

Se puede concluir que, en general, las arquitecturas con mejor rendimiento son la de K8s + KubeEdge y la de k3s + KubeEdge. La peor arquitectura para nodos con pocas o ninguna limitación de recursos es la de k3s *standalone*, mientras que la peor en el caso de un nodo con recursos limitados es la de K8s + OpenYurt.

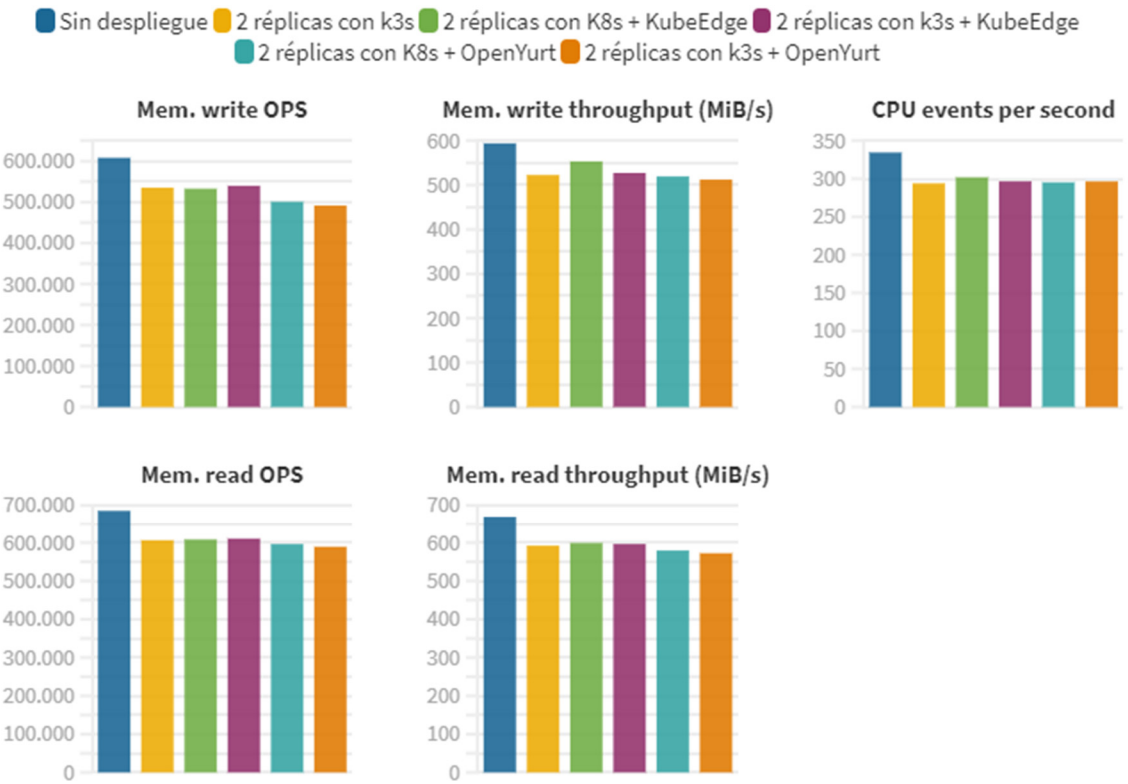

Figura 22. Rendimiento de memoria y CPU con Sysbench para el nodo Raspberry Pi 3.

Figura 23. Rendimiento de disco con FIO para el nodo Raspberry Pi 3.

6.4.3. Capacidades adicionales

Las capacidades adicionales específicas para casos de uso *edge* que ofrece cada distribución serían las siguientes:

- La instalación de k3s es compatible con dispositivos de recursos limitados y soporta ARM64 y ARM7.
- OpenYurt también es compatible con dispositivos con recursos limitados y permite, además, gestionar dispositivos distribuidos geográficamente en diferentes localizaciones *edge*.
- KubeEdge permite despliegues en dispositivos con recursos limitados, ofrece autonomía del *edge* para cuando existen inestabilidades en la conexión entre el *edge* y el *cloud*, y permite distribuir los recursos computaciones en diferentes localizaciones *edge*.

De las 3 distribuciones, KubeEdge es la que más capacidades adicionales ofrece para casos de uso de *edge computing*. En el otro extremo, k3s no ofrece demasiados añadidos para *edge* frente a una versión *vanilla* K8s, más allá de consumir menos recursos.

Teniendo en cuenta que tanto KubeEdge como OpenYurt necesitarían un despliegue de k3s o K8s por debajo, se podría pensar que el despliegue con k3s + KubeEdge/OpenYurt aporta ventajas frente a K8s + KubeEdge/Openyurt, pero no es el caso. Esto es debido a que únicamente se diferencian en el despliegue del nodo *cloud*, que es el que menos limitaciones de recursos computacionales y rendimiento presenta en un entorno real. Desplegar una versión ligera en un nodo que no se ve apenas limitado no ofrece ninguna ventaja provechosa.

6.4.4. Resumen de la selección

En base a lo descrito en los anteriores apartados, la distribución óptima para un despliegue de K8s en el *edge* es **KubeEdge**, tratándose de la distribución con mejores resultados en todos los ámbitos: madurez, utilización de recursos, rendimiento y capacidades adicionales. Dado que necesita un despliegue de K8s por debajo, se ha optado por utilizar K8s frente a k3s, principalmente por su mayor madurez y estabilidad. En la Tabla 2 se resume la selección realizada.

De esta forma, en el nodo *cloud* se tendrá un despliegue base de K8s y se desplegará sobre este el *CloudCore* de KubeEdge. En el nodo *edge*, en cambio, únicamente hará falta desplegar el componente *EdgeCore* de KubeEdge.

Tabla 2. Tabla resumen de selección de alternativas.

	Madurez	Utilización de recursos	Rendimiento	Capacidades *edge*	
	20%	20%	30%	30%	
K8s + KubeEdge	8	9	8	9	**8,5**
k3s + KubeEdge	4,5	9	8	10	8,1
K8s + OpenYurt	6,5	6	6	6	6,1
k3s + OpenYurt	3	6	7	6	5,7
Standalone k3s	3	5	6	3	4,3

7. Descripción del despliegue

Como prueba de concepto (PoC, *Proof of Concept*) de despliegue de *edge computing* se ha diseñado una arquitectura de microservicios que aprovecha algunas de las características específicas que ofrece KubeEdge, como los gemelos digitales y la autonomía del *edge*.

El diseño de la PoC se puede dividir en dos aplicaciones: la gestión del gemelo digital de un dispositivo y la monitorización y cálculo de estadísticas de un sensor IoT. Dado que en la práctica el despliegue no se ha dado en un entorno real de producción de industria 4.0, se han tenido que simular tanto el dispositivo como los datos generados por el sensor.

El esquema general es el mostrado en la Figura 24. Está formado por 3 nodos: el nodo A, situado en el CPD de las oficinas centrales en el *cloud*, y los nodos B y C, situados en el CPD de las oficinas de planta de la fábrica, en el *edge*. En la práctica, los 3 nodos han sido desplegados en el laboratorio del grupo de investigación I2T, dentro de la misma red. Los nodos utilizados son los utilizados previamente para las pruebas de rendimiento del apartado Análisis de alternativas, descritos en el subapartado 6.4.2:

- El **nodo A** es una máquina virtual alojada en el nodo OpenStack del laboratorio, a la que se le han asignado 8 cores, 16 GB de RAM y 200 GB de almacenamiento. Estas características son similares a las que podría tener un nodo *cloud* en un entorno de producción.
- El **nodo B** es una Raspberry Pi 3 con 4 cores, 1 GB de RAM y 8 GB de almacenamiento de tarjeta microSD. Dado que es un nodo con unos recursos muy limitados, su óptima utilización es de gran importancia.
- El **nodo C** es un servidor Supermicro con 12 cores, 64 GB de RAM y 477 GB de almacenamiento. Este servidor sirve como réplica o referencia a lo que se podría desplegar en un entorno real en el *edge*, cuando no existen excesivas limitaciones económicas, de movilidad o de espacio.

Cada uno de los hexágonos mostrados en la Figura 24 representa un microservicio. Aquellos de color azul forman parte del proceso de gestión de gemelos digitales, mientras que los de color verde forman parte del flujo para la monitorización y procesado de datos telemétricos de un sensor IoT. Los dos hexágonos de color naranja son utilizados en ambos flujos y representan dos MQTT *broker*, desplegados en cada uno de los nodos *edge*.

El despliegue de los microservicios se gestiona de forma centralizada desde el nodo A, en el *cloud*, a través de la herramienta CLI de K8s, *kubectl*. Estos microservicios se despliegan como *pods* en los nodos B y C, más concretamente, como contenedores Docker. En el nodo C, el más potente entre los dos del edge, se han desplegado ambas

aplicaciones, mientras que en el nodo B, de recursos mucho más limitados, únicamente se han desplegado los microservicios correspondientes a la gestión de un gemelo digital.

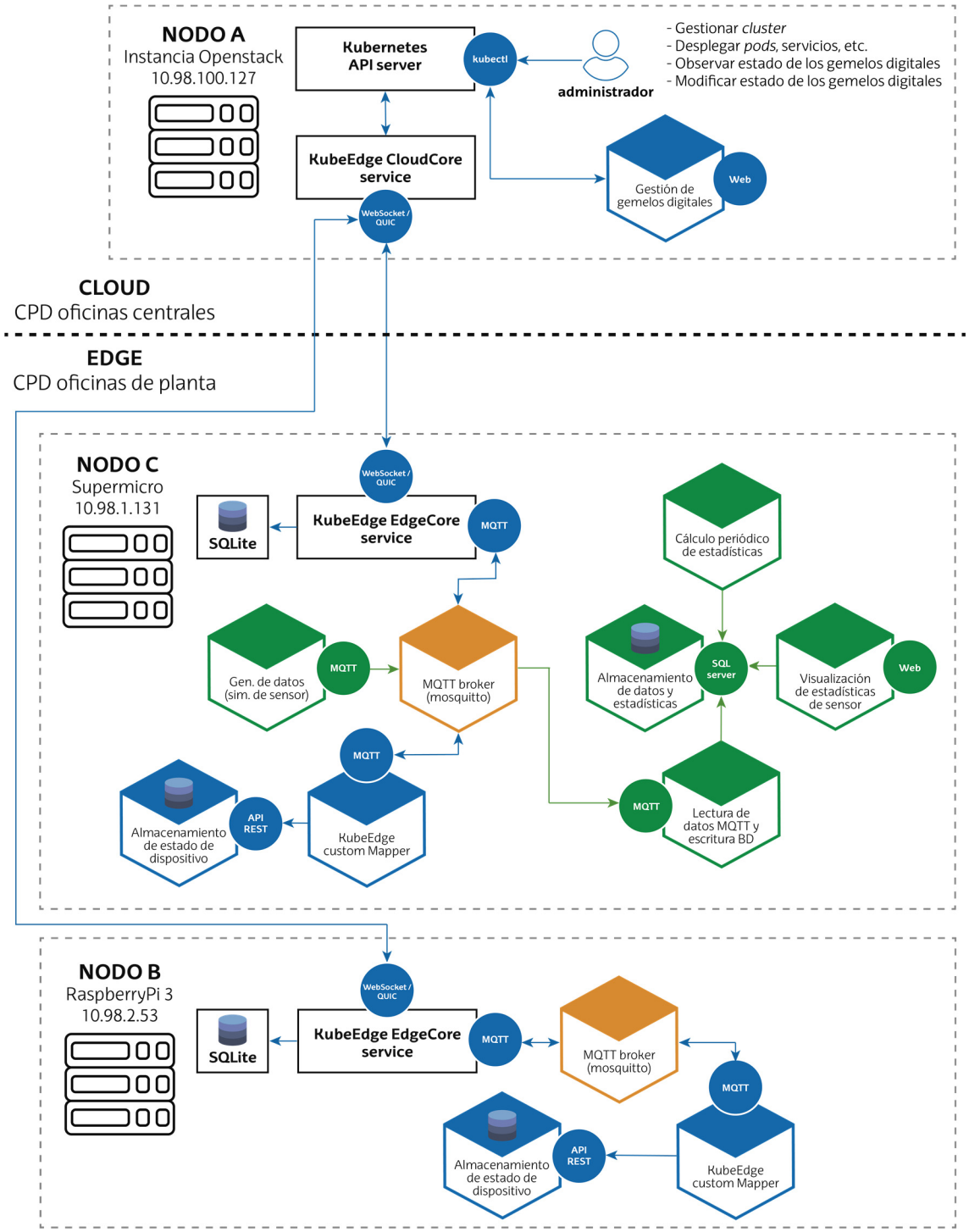

Figura 24. Esquema general de la arquitectura de microservicios del despliegue.

7.1. Descripción de los microservicios

El despliegue se compone de 9 microservicios diferenciados: 3 que forman parte de la aplicación de gestión de un gemelo digital, otros 5 que forman parte de la aplicación de monitorización y procesado de datos telemétricos de un sensor IoT, y el MQTT *broker*. Como ya se ha comentado, el MQTT *broker* forma parte del flujo de ambas aplicaciones.

A continuación, se describen ambos flujos y cada uno de los microservicios que componen las aplicaciones.

7.1.1. Gestión de gemelos digitales

La gestión de los gemelos digitales se basa en el servicio que ofrece KubeEdge por medio de los componentes *DeviceController* en el *cloud* y *DeviceTwin* en el *edge*. Cuando un dispositivo se conecta al nodo *edge*, el *DeviceTwin* genera de forma local en el *edge* un gemelo digital. Este gemelo digital no es más que el almacenamiento del estado del dispositivo (variables que definen el estado, como si está o no encendido o valores numéricos de algún sensor interno) en forma de pares clave-valor en una base de datos. Este estado se envía posteriormente al *DeviceController* del nodo *cloud*, quien almacena los datos en la base de datos de K8s, *etcd*. El servicio *EdgeCore* de KubeEdge se encarga de forma interna de generar el gemelo digital y de enviarle al servicio de KubeEdge del *cloud* la información. Desde el *cloud* se pueden configurar aquellos datos que sean de lectura y escritura, y leer los que sean solo de lectura, mediante la herramienta CLI de K8s, *kubectl*.

Los dispositivos se pueden conectar a un nodo *edge* por medio de múltiples protocolos. En un entorno industrial estos son, generalmente, Bluetooth, Modbus u OPC. KubeEdge ofrece unos servicios de mapeo específicos llamados *Mapper* tanto para Bluetooth como para Modbus, pero permite, además, diseñar *Mappers* personalizados para otros protocolos de comunicación que no soporta de forma nativa. Dado que en nuestro caso el dispositivo está simulado, se hará uso de un *Mapper* personalizado.

Para el despliegue de esta aplicación se ha utilizado de base el proyecto de ejemplo proporcionado por KubeEdge de control de un dispositivo que actúa como un contador. En la Figura 25 se muestra un esquema general de su funcionamiento. La aplicación se basa en el despliegue de 3 microservicios, los dos primeros desplegados en el nodo *edge* y el tercero desplegado en el nodo *cloud*:

— Microservicio de **almacenamiento del estado simulado** del contador. Almacena 2 variables, una que registra el estado del contador ON/OFF y otra que registra el valor actual del contador. La variable que define si el contador está o no encendido es de lectura y escritura, mientras que la del valor del contador es solo de lectura. Cuando el contador está encendido, este aumenta de valor cada segundo. Cuando está apagado, deja de contar y resetea el valor del contador a 0.

— Microservicio de **mapeo de información** entre el dispositivo simulado y el proceso *EdgeCore* de KubeEdge. Es el servicio de *Mapper* personalizado que se ha mencionado anteriormente. Se encarga de enviar al MQTT *broker* los datos del contador en un formato apropiado para que los procese KubeEdge y viceversa. Concretamente,
— Microservicio de **servidor web** para gestionar el estado del gemelo digital. Este será el único microservicio desplegado en el nodo A del *cloud*. Se trata de una interfaz web de usuario que actúa de *frontend* con una aplicación de *backend* que gestiona las peticiones realizadas por el usuario a través de la interfaz y las traduce a llamadas a la API de K8s. El servicio se expone en el puerto 80 del nodo en el que se despliega, que en este caso es el nodo A. Mediante dicha interfaz web se podrá modificar el estado del contador (ponerlo en ON o en OFF), así como visualizar el valor actual del contador.

Adicionalmente, se hace uso de un MQTT *broker* desplegado en el nodo *edge* para comunicar el microservicio del *Mapper* personalizado y el servicio *EdgeCore* de KubeEdge. Este broker se despliega de forma automática junto a la instalación de KubeEdge mediante el paquete de Ubuntu *mosquitto*. Este paquete es la implementación para Ubuntu del MQTT *broker* Eclipse Mosquitto, desarrollado por la Fundación Eclipse [36].

Existe un tema MQTT para que el *Mapper* del nodo *edge* publique las actualizaciones del valor del contador y KubeEdge las lea, y otro tema MQTT para que KubeEdge publique los cambios de configuración en el estado del contador aplicados a través de la interfaz web del nodo *cloud* y el *Mapper* las lea para hacérselas llegar al microservicio del contador. Cuando se pierde la conectividad entre ambos nodos, el contador seguirá aumentando su valor en el *edge*, pero el nodo *cloud* no obtendrá la actualización hasta que se reestablezca la conexión. Asimismo, los combios de configuración realizados en el *cloud* a través de la interfaz web no tendrán efecto sin conexión entre nodos.

En la Figura 26, la Figura 27 y la Figura 28 se muestran un diagrama de casos de uso y dos de secuencia, respectivamente, que permiten visualizar de forma más esquemática el funcionamiento de la aplicación y su flujo de datos.

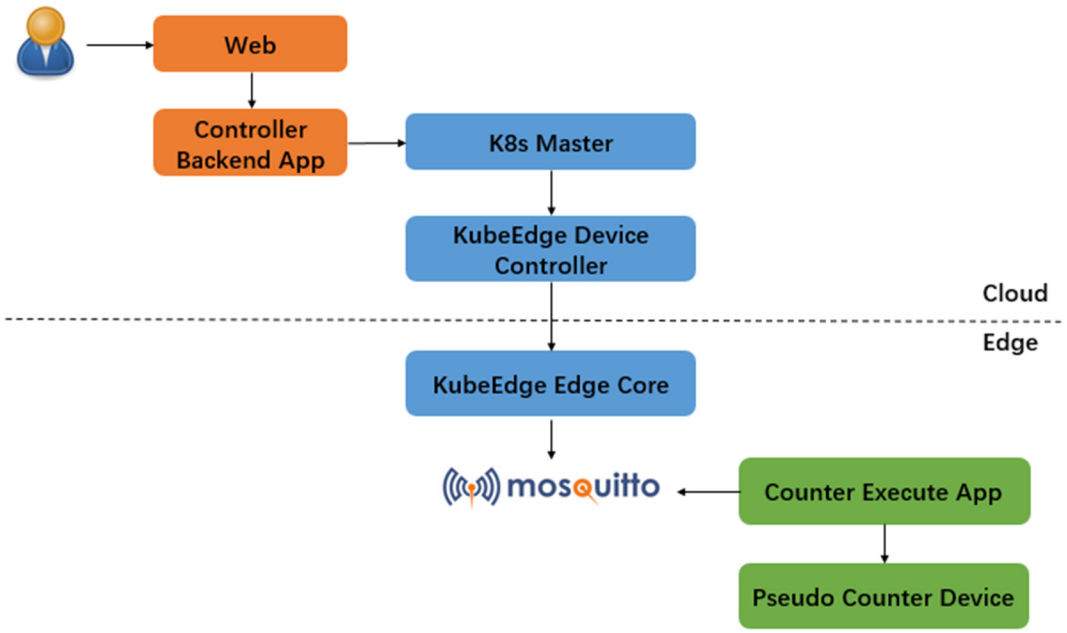

Figura 25. Esquema general de la aplicación de simulación de un contador [37].

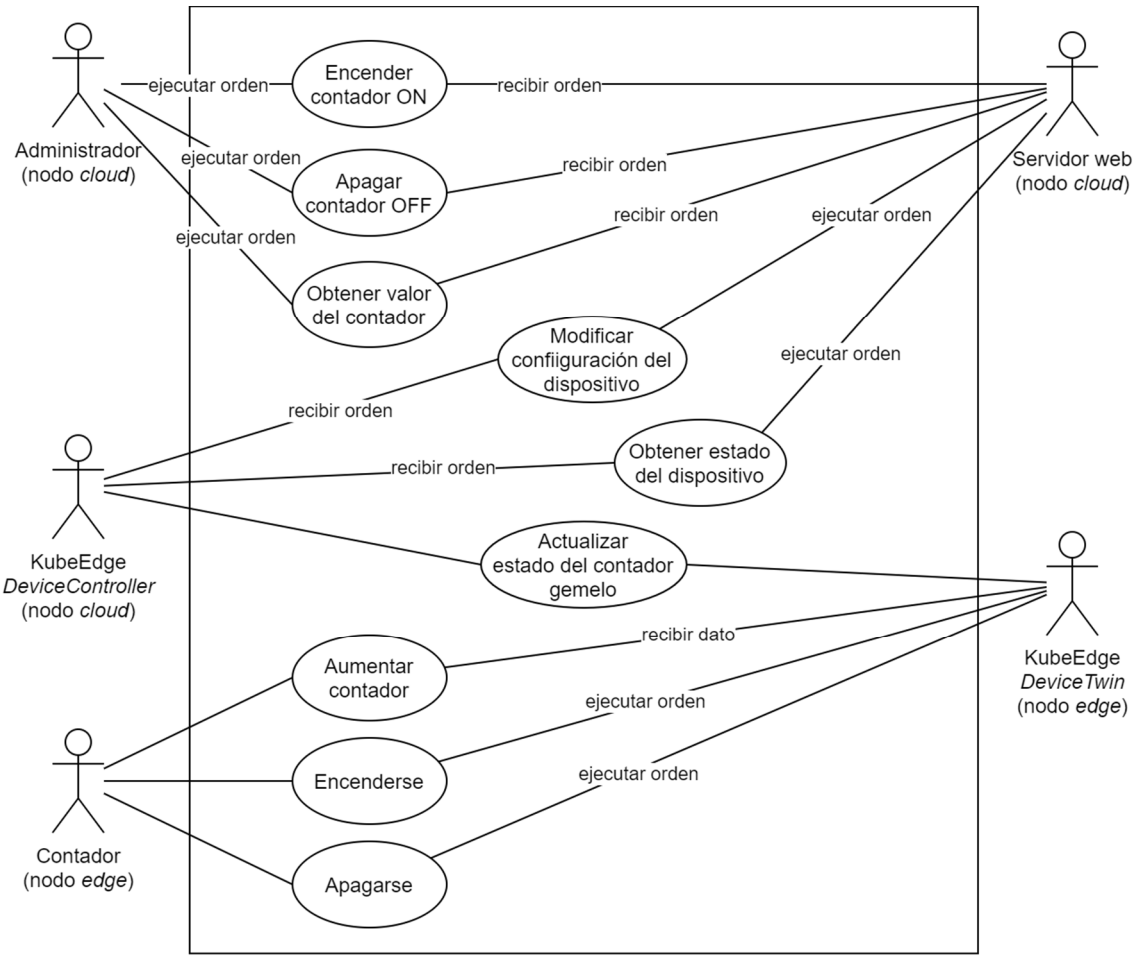

Figura 26. Diagrama de casos de uso de la aplicación de gemelo digital de un contador.

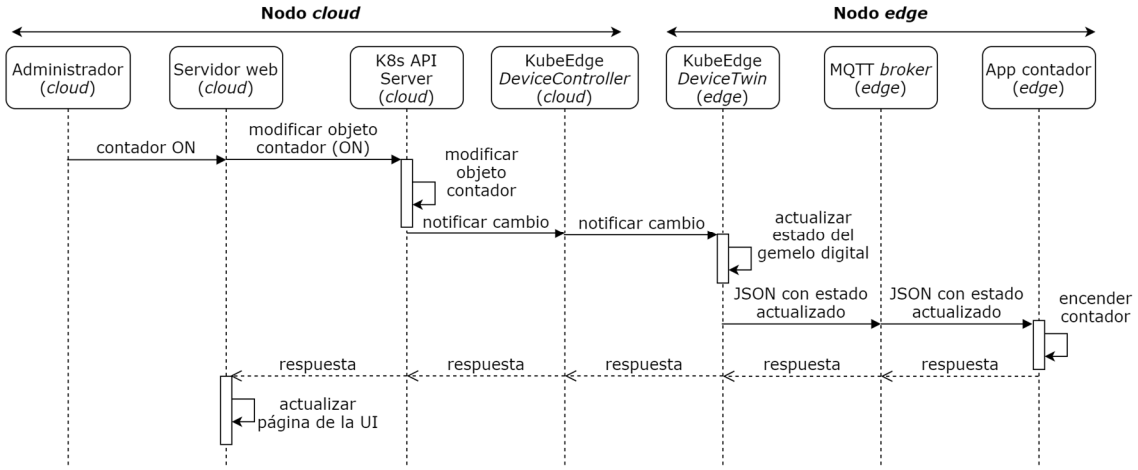

Figura 27. Diagrama de secuencia del proceso de encendido del contador.

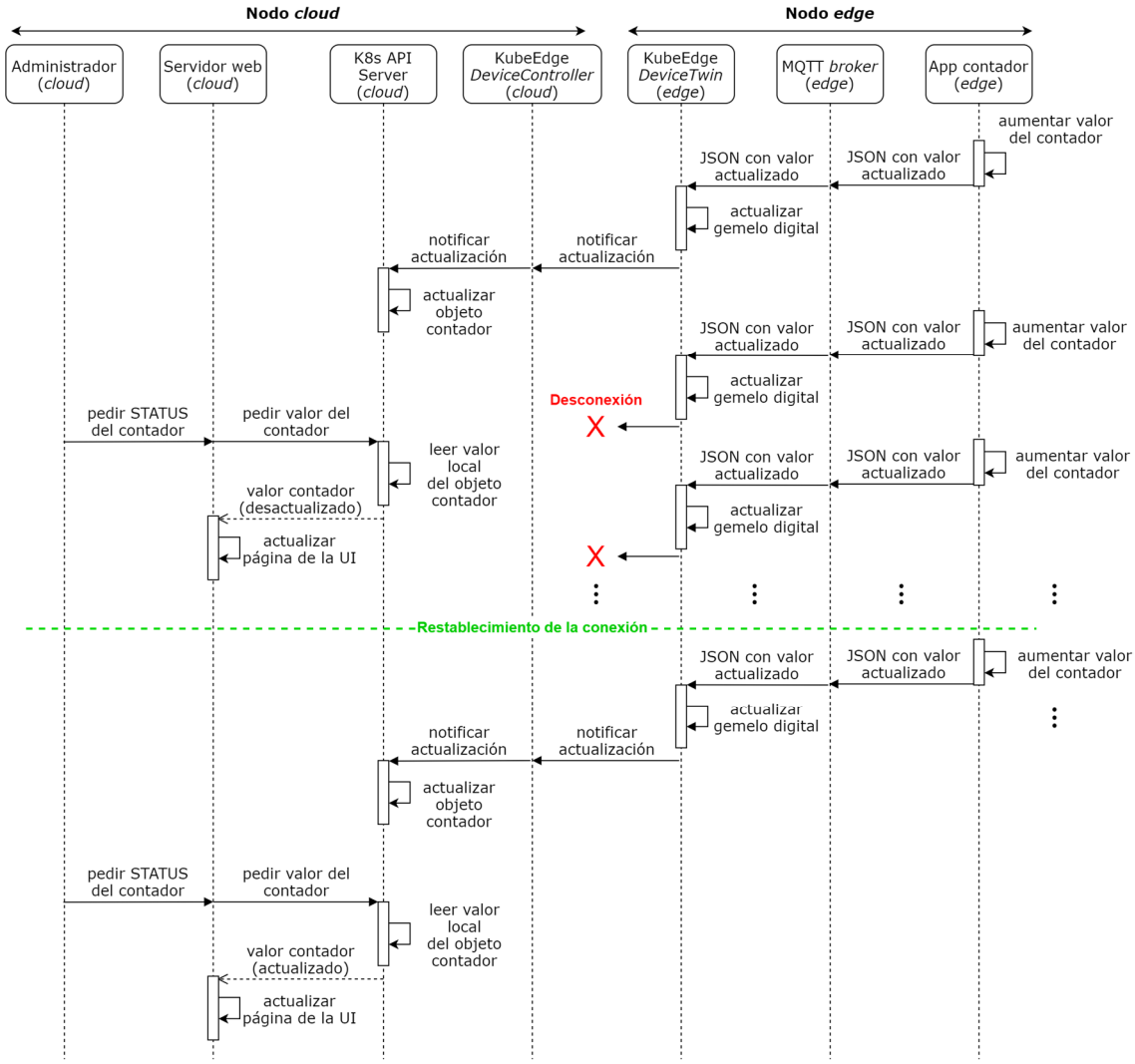

Figura 28. Diagrama de secuencia del proceso de actualización del valor del contador.

7.1.2. Datos telemétricos de un sensor IoT

Esta aplicación pretende demostrar el caso de uso de monitorización de una planta industrial por medio del procesado de datos generados por los sensores IoT repartidos en ella. En un entorno real, el procesado de los datos puede llegar a ser muy complejo, utilizando técnicas de *machine learning* y *big data*. Sin embargo, el objetivo de este proyecto no es estudiar el uso de dichas técnicas, por lo que el procesado de los datos se ha simplificado para poder centrarse en la demostración del propio despliegue de los microservicios.

Dado que para el proyecto no se disponía de sensores físicos para su recolección continua de datos, se ha simulado esta generación mediante un microservicio adicional paralelo a la parte del procesado. Así, la aplicación se compone de 5 microservicios:

- Microservicio de **generación de datos** telemétricos simulados de un sensor IoT.
- Microservicio de **lectura de datos generados** y escritura en una BD externa.
- Microservicio de procesado de datos y **cálculo de estadísticas** de esos datos.
- Microservicio de **almacenamiento de datos** simulados y sus estadísticas calculadas.
- Microservicio de **visualización de datos** simulados y sus estadísticas.

Al igual que en el caso anterior, se tendrá desplegado en el nodo *edge* un MQTT *broker* al que el simulador del sensor enviará los datos generados y del que el microservicio de lectura de datos leerá para obtener los datos a almacenar en una BD externa.

En la Figura 29 se muestra un diagrama simplificado del flujo de la aplicación completa. A continuación se describirán con algo más de detalle los microservicios involucrados. Para desplegar los 4 primeros se han creado imágenes de Docker personalizadas para cada aplicación ejecutada por el microservicio. En el Anexo II se incluyen los ficheros ejecutables y de configuración que se han utilizado para desplegar cada uno, así como el contenido de los ficheros Dockerfile utilizados para crear las imágenes de contenedores Docker personalizadas.

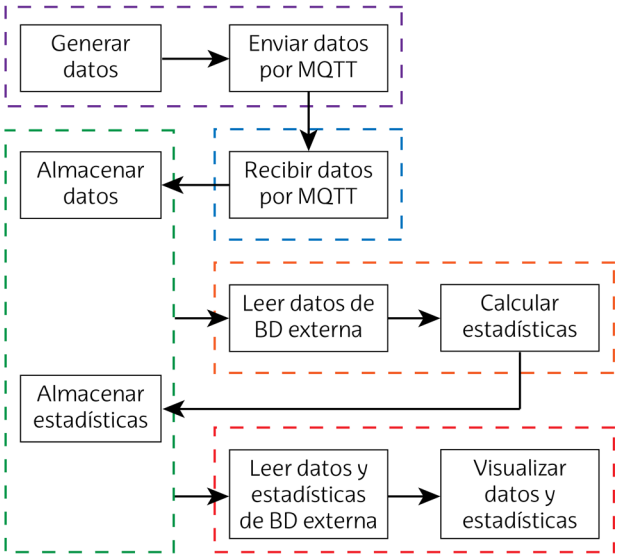

Figura 29. Diagrama de flujo desde la generación de datos hasta la visualización.

7.1.2.1. Microservicio de generación de datos

Este microservicio se basa en una simple aplicación desarrollada en Python que genera 6 tipos de señal, como las que podrían generar varios sensores IoT:

- Seno de periodo 8 segundos: $y(t) = \sin(\pi t/4)$
- Seno de periodo más largo de 24 segundos: $y(t) = \sin(\pi t/12)$
- Señal triangular de periodo 6 segundos: $y(t) = \sum_n 2\Delta\left(\frac{t-n}{3}\right) - 1,\ n = 0, 6, 12, 18 \ldots$
- Señal creciente de forma lineal con el tiempo: $\mathrm{y(t) = t/1000}$
- Señal constante en el tiempo: $y(t) = 10$
- Señal aleatoria

Para generar los senos y la señal triangular se ha hecho uso de la librería numpy de Python.

La frecuencia de generación de todas ellas es de una muestra por segundo. Las muestras generadas cada segundo las recoge en un JSON, junto a un sello de tiempo, y las publica en el MQTT *broker* bajo el tema "mocksensor". Este proceso es iterativo de forma infinita, hasta que se termine de forma forzosa la aplicación.

7.1.2.2. Microservicio de lectura de datos generados

Este microservicio se encarga de suscribirse al tema MQTT "mocksensor", utilizado por el microservicio anterior para publicar los datos, y almacenar los datos recibidos en una base de datos externa. Para evitar tener una base de datos de grandes dimensiones, únicamente se almacenarán las últimas 100 entradas, correspondientes a las últimas 100 muestras generadas de cada señal.

7.1.2.3. Microservicio de cálculo de estadísticas

El microservicio de cálculo de estadísticas utiliza los datos generados por el microservicio de generación de datos que se han almacenado previamente en una base de datos y extrae los valores máximos, mínimos, la media y la desviación típica. Calcula las estadísticas cada 10 segundos, haciendo uso de las 100 muestras de las señales generadas almacenadas en ese instante en la base de datos. Al igual que para el caso anterior, para minimizar el tamaño de la base de datos, se retienen únicamente las 10 últimas entradas de estadísticas calculadas.

7.1.2.4. Microservicio de almacenamiento de datos y estadísticas

El microservicio de almacenamiento representa la base de datos externa que utilizan los anteriores microservicios mencionados. La base de datos se compone de 5 tablas, una para los datos del sensor y otras 4 para los datos estadísticos calculados, como muestra la Figura 30.

En la tabla *sensor_data* se almacenarán los valores de las 100 últimas muestras generadas de los 6 tipos de señal. En las tablas *sensor_data_stats_max*, *sensor_data_stats_min*, *sensor_data_stats_mean* y *sensor_data_stats_stdev* se almacenan las 10 últimas estadísticas calculadas de valor máximo, valor mínimo, valor medio y desviación estándar de las señales generadas y almacenadas en la tabla *sensor_data*.

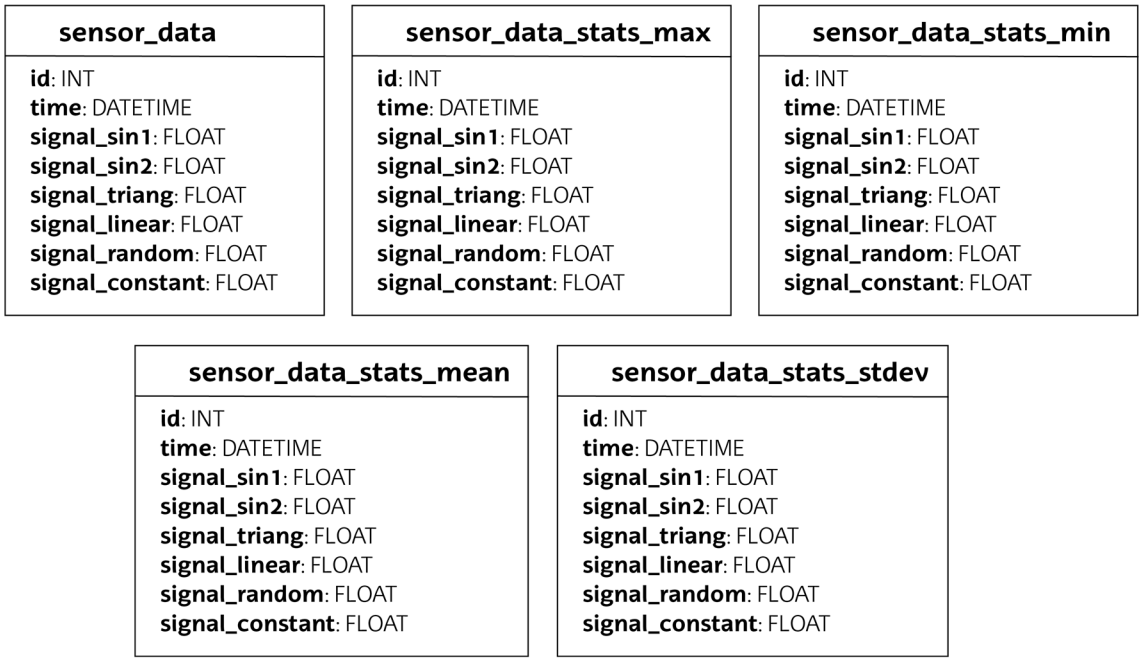

Figura 30. Diagrama de tablas de la BD del microservicio de almacenamiento de datos.

7.1.2.5. Microservicio de visualización de datos

El microservicio de visualización de datos hace uso de las 5 tablas mencionadas del microservicio de almacenamiento y las visualiza por medio de la herramienta Grafana.

Grafana es un software de código abierto de análisis y visualización de datos. Permite consultar, visualizar y explorar múltiples métricas basadas en tiempo, independientemente de su origen o lugar de almacenamiento. Además, permite configurar alertas sobre dichas métricas. Esta visualización de datos se hace a través de una interfaz web de usuario en la que se pueden configurar varios paneles sobre los que representar los datos [38].

En el caso de la aplicación desplegada, se representarán en un panel las 6 señales generadas y en otro panel las estadísticas.

8. Análisis de los resultados

8.1. Gestión de gemelo digital

Para analizar el funcionamiento del despliegue del gemelo digital, se han hecho dos comprobaciones:

1. Por un lado, se ha probado cómo desde el *cloud* se puede modificar una variable de configuración del dispositivo simulado y obtener el contenido de otra variable, accediendo al servicio web desplegado.
2. Por otro lado, se ha probado la autonomía que adquiere el *edge* para seguir funcionando cuando pierde la conectividad con el *cloud*.

En la Figura 31 se muestra el valor del contador en 3 puntos: en los *logs* que genera el dispositivo contador simulado del *edge*, la interfaz del servicio web desplegado en el nodo *cloud* y lo almacenado en el servidor API de K8s del nodo *cloud*. En este estado inicial, existe conectividad entre ambos nodos y, como se puede observar, en los 3 sitios se muestra el mismo valor del contador.

Una vez se pierde la conectividad entre ambos nodos, el dispositivo contador sigue generando los *logs* en el contenedor desplegado en el *edge*. Sin embargo, esos datos no se actualizan en el nodo *cloud*, como se observa en la Figura 32. Una vez esta conexión se restableciese, los valores se volverían a sincronizar.

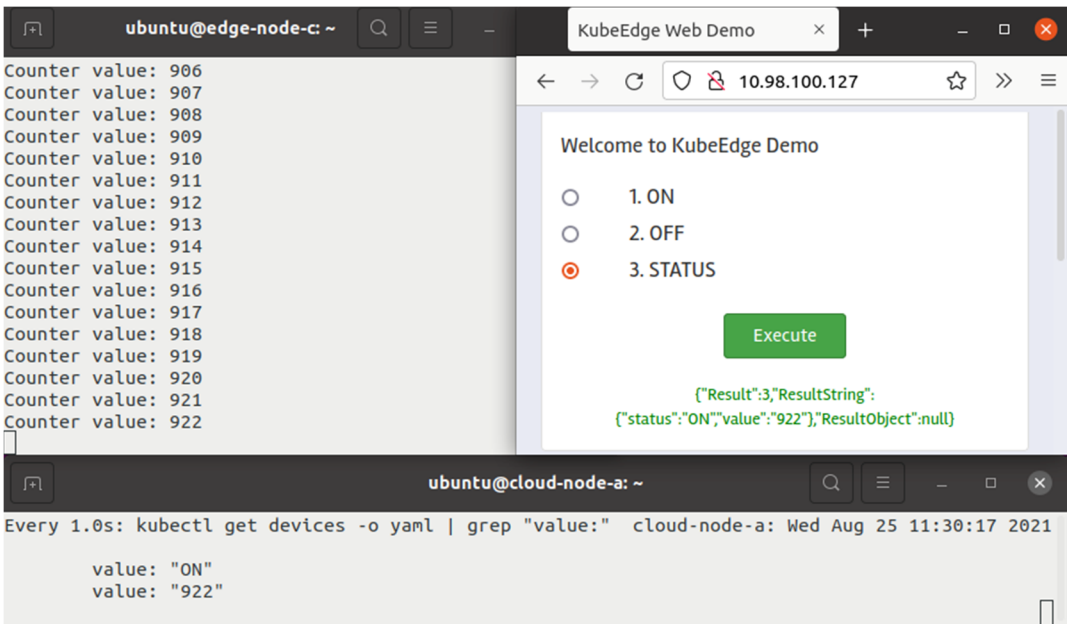

Figura 31. Comparativa del valor del contador en el *edge* (arriba a la izquierda) con el valor almacenado en el *cloud* (abajo) y lo mostrado por el servidor web del *cloud* (arriba a la derecha) cuando hay conectividad entre nodos.

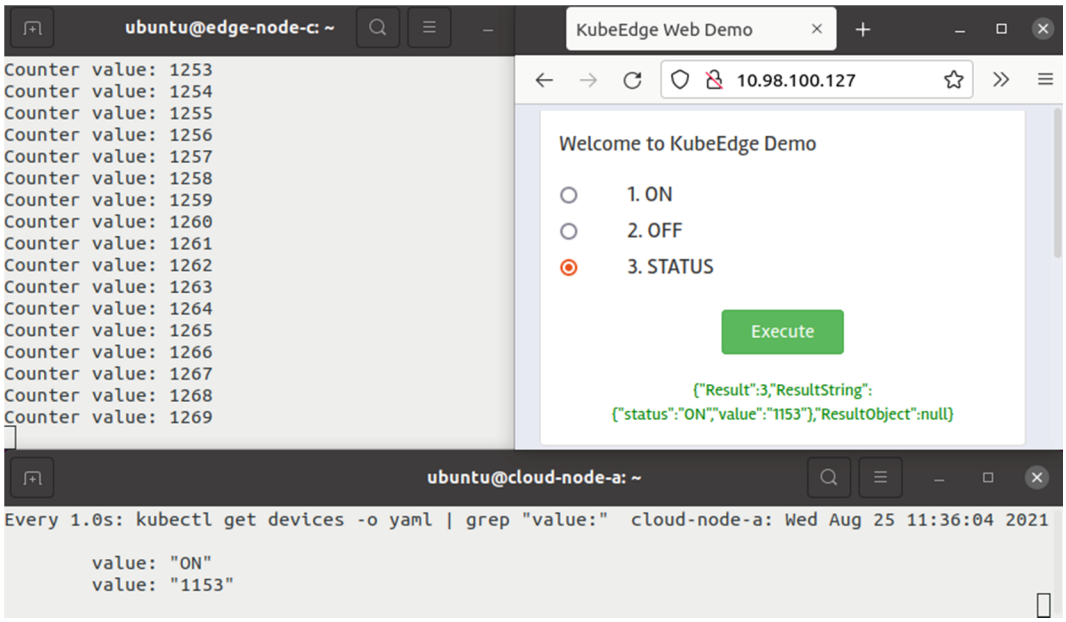

Figura 32. Comparativa del valor del contador en el *edge* (arriba a la izquierda) con el valor almacenado en el *cloud* (abajo) y lo mostrado por el servidor web del *cloud* (arriba a la derecha) cuando no hay conectividad entre nodos.

8.2. Datos telemétricos de un sensor IoT

Para analizar los resultados de esta segunda aplicación, se ha analizado primero que cada microservicio realizaba las acciones que debía.

Primero, se ha observado que el microservicio de generación de datos generaba correctamente las muestras de las señales, una muestra de cada señal por segundo y que las publicaba por MQTT. Para comprobar esto, se ha hecho una prueba desde un nodo de la misma red suscribiéndose al tema "mocksensor" mediante la ejecución del comando `mosquitto_sub`.

A continuación, se ha comprobado que el microservicio de lectura de datos leía correctamente los datos y los almacenaba en la base de datos. Como se puede observar en la Figura 33, los datos se almacenan correctamente.

```
mysql> select * from sensor_data;
+------+---------------------+-------------+-------------+--------------+--------------+--------------+-----------------+
| id   | time                | signal_sin1 | signal_sin2 | signal_triang | signal_linear | signal_random | signal_constant |
+------+---------------------+-------------+-------------+--------------+--------------+--------------+-----------------+
| 2269 | 2021-08-22 07:31:44 |  3.67394e-16 |  1.22465e-16 |           -1 |        0.732 |     0.928121 |              10 |
| 2270 | 2021-08-22 07:31:45 |    -0.707107 |    -0.258819 |    -0.333333 |        0.733 |     0.843746 |              10 |
| 2271 | 2021-08-22 07:31:46 |           -1 |        -0.5 |     0.333333 |        0.734 |        0.748 |              10 |
| 2272 | 2021-08-22 07:31:47 |    -0.707107 |    -0.707107 |            1 |        0.735 |     0.372564 |              10 |
| 2273 | 2021-08-22 07:31:48 |            0 |    -0.866025 |     0.333333 |        0.736 |     0.594447 |              10 |
| 2274 | 2021-08-22 07:31:49 |     0.707107 |    -0.965926 |    -0.333333 |        0.737 |     0.827608 |              10 |
| 2275 | 2021-08-22 07:31:50 |            1 |           -1 |           -1 |        0.738 |     0.114081 |              10 |
| 2276 | 2021-08-22 07:31:51 |     0.707107 |    -0.965926 |    -0.333333 |        0.739 |     0.596691 |              10 |
| 2277 | 2021-08-22 07:31:52 |  1.22465e-16 |    -0.866025 |     0.333333 |         0.74 |     0.464458 |              10 |
| 2278 | 2021-08-22 07:31:53 |    -0.707107 |    -0.707107 |            1 |        0.741 |     0.739839 |              10 |
| 2279 | 2021-08-22 07:31:54 |           -1 |        -0.5 |     0.333333 |        0.742 |     0.838337 |              10 |
```

Figura 33. Listado de datos simulados del sensor almacenados en la BD.

Para comprobar que el microservicio de cálculo de estadísticas generaba las estadísticas y las almacenaba en la base de datos se han observado en la base de datos las tablas correspondientes. La Figura 34, Figura 35, Figura 36 y Figura 37 muestran el contenido de estas tablas. Se puede comprobar también cómo únicamente se almacenan los últimos 10 valores, como se había diseñado.

```
mysql> select * from sensor_data_stats_max;
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
| id  | time                | signal_sin1 | signal_sin2 | signal_triang | signal_linear | signal_random | signal_constant |
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
| 230 | 2021-08-22 07:31:54 |           1 |           1 |             1 |        0.742 |      0.978367 |              10 |
| 231 | 2021-08-22 07:32:04 |           1 |           1 |             1 |        0.752 |      0.978367 |              10 |
| 232 | 2021-08-22 07:32:14 |           1 |           1 |             1 |        0.762 |       0.99447 |              10 |
| 233 | 2021-08-22 07:32:24 |           1 |           1 |             1 |        0.772 |       0.99447 |              10 |
| 234 | 2021-08-22 07:32:35 |           1 |           1 |             1 |        0.782 |       0.99447 |              10 |
| 235 | 2021-08-22 07:32:45 |           1 |           1 |             1 |        0.792 |       0.99447 |              10 |
| 236 | 2021-08-22 07:32:55 |           1 |           1 |             1 |        0.802 |       0.99447 |              10 |
| 237 | 2021-08-22 07:33:05 |           1 |           1 |             1 |        0.812 |       0.99447 |              10 |
| 238 | 2021-08-22 07:33:15 |           1 |           1 |             1 |        0.822 |       0.99447 |              10 |
| 239 | 2021-08-22 07:33:25 |           1 |           1 |             1 |        0.831 |      0.994537 |              10 |
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
10 rows in set (0.00 sec)
```

Figura 34. Listado de valores máximos de los datos almacenados en la BD.

```
mysql> mysql> select * from sensor_data_stats_min;
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
| id  | time                | signal_sin1 | signal_sin2 | signal_triang | signal_linear | signal_random | signal_constant |
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
| 230 | 2021-08-22 07:31:54 |          -1 |          -1 |            -1 |        0.643 |    0.00262213 |              10 |
| 231 | 2021-08-22 07:32:04 |          -1 |          -1 |            -1 |        0.653 |    0.00262213 |              10 |
| 232 | 2021-08-22 07:32:14 |          -1 |          -1 |            -1 |        0.663 |    0.00262213 |              10 |
| 233 | 2021-08-22 07:32:24 |          -1 |          -1 |            -1 |        0.673 |    0.00262213 |              10 |
| 234 | 2021-08-22 07:32:35 |          -1 |          -1 |            -1 |        0.683 |    0.00262213 |              10 |
| 235 | 2021-08-22 07:32:45 |          -1 |          -1 |            -1 |        0.693 |    0.00262213 |              10 |
| 236 | 2021-08-22 07:32:55 |          -1 |          -1 |            -1 |        0.703 |    0.00962984 |              10 |
| 237 | 2021-08-22 07:33:05 |          -1 |          -1 |            -1 |        0.713 |    0.00962984 |              10 |
| 238 | 2021-08-22 07:33:15 |          -1 |          -1 |            -1 |        0.723 |    0.00962984 |              10 |
| 239 | 2021-08-22 07:33:25 |          -1 |          -1 |            -1 |        0.732 |    0.00962984 |              10 |
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
10 rows in set (0.00 sec)
```

Figura 35. Listado de valores mínimos de los datos almacenados en la BD.

```
mysql> select * from sensor_data_stats_mean;
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
| id  | time                | signal_sin1 | signal_sin2 | signal_triang | signal_linear | signal_random | signal_constant |
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
| 230 | 2021-08-22 07:31:54 |       -0.01 |  -0.0303906 |     0.0133333 |       0.6925 |       0.49422 |              10 |
| 231 | 2021-08-22 07:32:04 |  -0.0241421 |   0.0379788 |    -0.0133333 |       0.7025 |      0.511298 |              10 |
| 232 | 2021-08-22 07:32:14 |        0.01 |  -0.0353906 |             0 |       0.7125 |      0.531762 |              10 |
| 233 | 2021-08-22 07:32:24 |   0.0241421 |   0.0233195 |     0.0133333 |       0.7225 |       0.53769 |              10 |
| 234 | 2021-08-22 07:32:35 |       -0.01 |      -0.005 |    -0.0133333 |       0.7325 |      0.561622 |              10 |
| 235 | 2021-08-22 07:32:45 |  -0.0241421 |  -0.0146593 |             0 |       0.7425 |       0.55611 |              10 |
| 236 | 2021-08-22 07:32:55 |        0.01 |   0.0303906 |     0.0133333 |       0.7525 |      0.592649 |              10 |
| 237 | 2021-08-22 07:33:05 |   0.0241421 |  -0.0379788 |    -0.0133333 |       0.7625 |      0.585456 |              10 |
| 238 | 2021-08-22 07:33:15 |       -0.01 |   0.0353906 |             0 |       0.7725 |      0.584211 |              10 |
| 239 | 2021-08-22 07:33:25 |  -0.0241421 |  -0.0146593 |             0 |       0.7815 |      0.569541 |              10 |
+-----+---------------------+-------------+-------------+--------------+--------------+---------------+-----------------+
10 rows in set (0.00 sec)
```

Figura 36. Listado de valores medios de los datos almacenados en la BD.

```
mysql> select * from sensor_data_stats_stdev;
+-----+---------------------+-------------+-------------+---------------+---------------+---------------+-----------------+
| id  | time                | signal_sin1 | signal_sin2 | signal_triang | signal_linear | signal_random | signal_constant |
+-----+---------------------+-------------+-------------+---------------+---------------+---------------+-----------------+
| 230 | 2021-08-22 07:31:54 |    0.710598 |    0.713086 |      0.639023 |     0.0290115 |       0.29215 |               0 |
| 231 | 2021-08-22 07:32:04 |    0.710255 |    0.721053 |      0.639023 |     0.0290115 |      0.291389 |               0 |
| 232 | 2021-08-22 07:32:14 |    0.710598 |    0.718147 |      0.646149 |     0.0290115 |       0.29899 |               0 |
| 233 | 2021-08-22 07:32:24 |    0.710255 |    0.707197 |      0.639023 |     0.0290115 |      0.305445 |               0 |
| 234 | 2021-08-22 07:32:35 |    0.710598 |    0.699072 |      0.639023 |     0.0290115 |       0.29854 |               0 |
| 235 | 2021-08-22 07:32:45 |    0.710255 |    0.702057 |      0.646149 |     0.0290115 |      0.304077 |               0 |
| 236 | 2021-08-22 07:32:55 |    0.710598 |    0.713086 |      0.639023 |     0.0290115 |      0.286917 |               0 |
| 237 | 2021-08-22 07:33:05 |    0.710255 |    0.721053 |      0.639023 |     0.0290115 |      0.290651 |               0 |
| 238 | 2021-08-22 07:33:15 |    0.710598 |    0.718147 |      0.646149 |     0.0290115 |      0.296052 |               0 |
| 239 | 2021-08-22 07:33:25 |    0.710255 |    0.702057 |      0.646149 |     0.0290115 |      0.304579 |               0 |
+-----+---------------------+-------------+-------------+---------------+---------------+---------------+-----------------+
10 rows in set (0.00 sec)
```

Figura 37. Listado de desviaciones típicas de los datos almacenados en la BD.

Por último, para comprobar el despliegue del microservicio de visualización de datos, se ha accedido al servicio web de visualización que ofrece la herramienta Grafana. Para ello, se ha accedido al puerto 3000 del nodo *edge*, que es donde se ha realizado la asociación de puertos para acceder al servicio del contenedor.

En la Figura 38 y la Figura 39 se muestran las señales y sus estadísticas representadas de forma gráfica. Como se puede observar, se dibujan todas las señales correctamente. Asimismo, las estadísticas calculadas son las esperadas para cada una de las señales. Se concluye, así, que el despliegue es satisfactorio y cumple con su cometido.

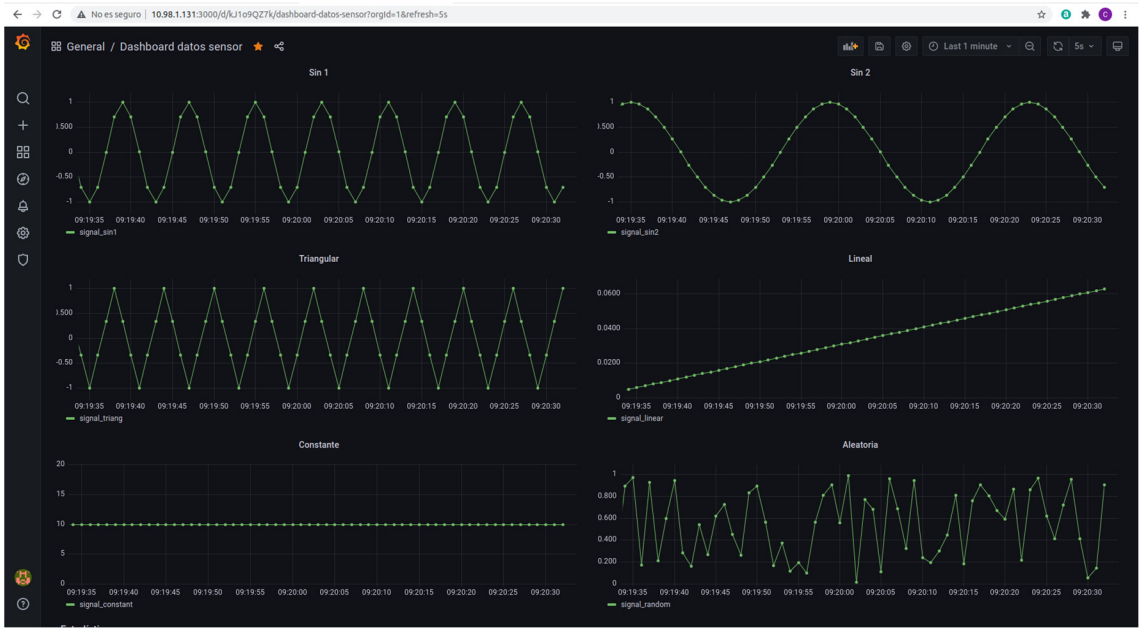

Figura 38. Visualización de las 6 señales simuladas y almacenadas en la BD.

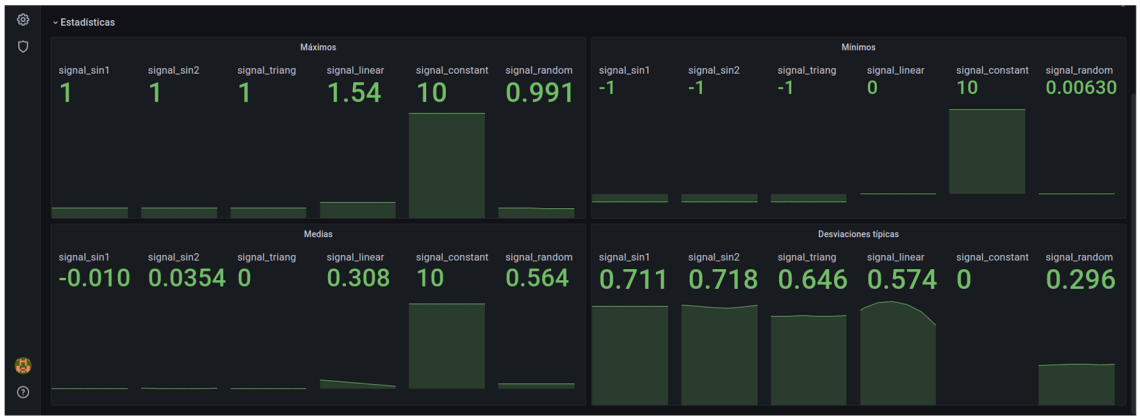

Figura 39. Visualización de las estadísticas de los datos almacenadas en la BD.

9. Planificación

Para la planificación del proyecto se ha asumido que este se desarrollaría entre el 1 de febrero y el 24 de septiembre. Entre esas fechas hay 165 días laborables, sin contar los fines de semana y los festivos. Los días que se han considerado festivos son aquellos oficialmente festivos en el País Vasco. Así, se han teniendo en cuenta los siguientes días festivos que caen entre semana:

- 19 de marzo, San José
- 1 de abril, Jueves Santo
- 2 de abril, Viernes Santo
- 5 de abril, Lunes de Pascua
- 3 de mayo, Día del Trabajo

Adicionalmente, se han asignado 14 días laborables de vacaciones, divididas en 2 periodos: del 6 al 9 de abril y del 23 de agosto al 3 de septiembre. Teniendo estas vacaciones en cuenta, el proyecto contiene 151 días laborables. Estos días, a 4 horas de trabajo al día como media jornada, equivalen a 604 horas de trabajo.

9.1. Descripción de tareas

El proyecto se ha dividido en 6 paquetes de tareas:

P.T.1. Definición del proyecto
P.T.2. Estudio y análisis de las tecnologías
P.T.3. Análisis de alternativas
P.T.4. Diseño de la prueba de concepto (PoC)
P.T.5. Despliegue de la PoC
P.T.6. Gestión del proyecto y documentación

Estos paquetes de tareas, así como las tareas que contiene cada uno se describen de forma breve en la Tabla 3. Asimismo, se muestra la duración de cada una de ellas en forma de Diagrama de Gantt en la Figura 40.

Tabla 3. Descripción de las tareas que forman el proyecto.

Código	Nombre de la tarea	Descripción de la tarea
P.T.1	**Definición del proyecto**	**Definición de las bases del proyecto**
T.1.1	Discusión y definición del tema	Definición de las bases del proyecto
T.1.2	Definición de alcance y objetivos	Definición del tipo de trabajo, sus objetivos específicos y el alcance
P.T.2	**Estudio y análisis de las tecnologías**	**Formación en los conceptos y técnicas necesarias para el desarrollo del proyecto**
T.2.1	Estudio de tecnologías de *edge computing*	Estudio de la normativa y arquitectura de la tecnología MEC. Introducción al *edge computing*
T.2.2	Estudio de tecnologías de orquestación (K8s)	Formación previa de las bases de las tecnologías de orquestación. Introducción a Kubernetes
T.2.3	Estudio de las necesidades de la industria 4.0	Estudio de necesidades específicas de la industria 4.0 (latencia, ancho de banda, etc.) aplicado a tecnologías *edge* (IoT, machine learning, protocolos de comunicación, etc.)
T.2.4	Análisis del estado del arte	Búsqueda de las distribuciones disponibles de Kubernetes para *edge computing*
P.T.3	**Análisis de alternativas**	**Seleccionar la alternativa más adecuada para llevar el proyecto a cabo entre las encontradas en el análisis del estado del arte.**
T.3.1	Diseño de las pruebas de rendimiento	Estudio de los KPI a medir en las pruebas y selección de herramientas a utilizar
T.3.2	Medidas de rendimiento	Ejecución de las pruebas de rendimiento
T.3.3	Análisis y selección de alternativas	Análisis y selección de las alternativas en base a las pruebas de rendimiento y demás criterios de selección
P.T.4	**Diseño de la prueba de concepto (PoC)**	**Diseño básico y de alto nivel de la arquitectura de despliegue**
T.4.1	Análisis de requerimientos	Analizar los requerimientos de despliegue
T.4.2	Diseño de la arquitectura	Diseñar la arquitectura, con sus componentes, nodos y aplicaciones necesarias
P.T.5	**Despliegue de la PoC**	**Implementación de la PoC**
T.5.1	Configuración del escenario	Configuración de los equipos, desarrollo de las aplicaciones y creación de contenedores
T.5.2	Despliegue de aplicaciones	Despliegue de las aplicaciones y los contenedores en los nodos del escenario
P.T.6	**Gestión del proyecto y documentación**	**Elaboración de la documentación del proyecto**
T.6.1	Documentación del proyecto	Generación de la documentar el proyecto completo
T.6.2	Revisión final de la documentación	Revisión de la documentación final
H.6.3	Entrega final	Hito de entrega final

9.2. Diagrama de Gantt

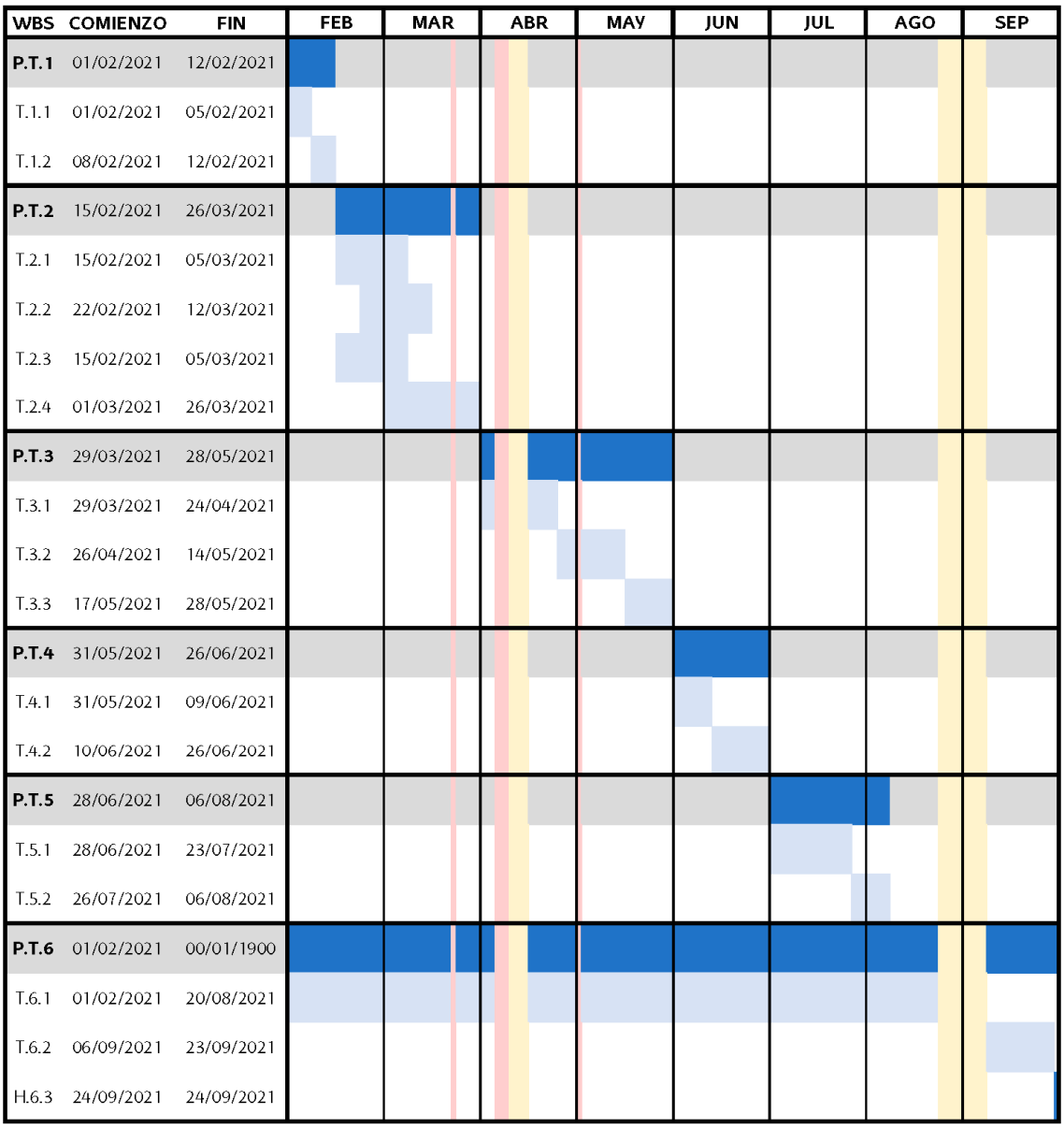

Figura 40. Diagrama de Gantt.

10. Descripción del presupuesto

En este apartado se definen los costes asociados a los recursos necesarios para llevar a cabo el proyecto. Estos costes se componen de costes directos e indirectos, así como un porcentaje dedicado a imprevistos (10%).

Por lo general, los costes directos se desglosan en 4 partidas: horas internas, amortizaciones, gastos y subcontrataciones. Sin embargo, en este proyecto no se ha presupuestado ninguna subcontratación. Las 3 partidas restantes se detallan en la Tabla 4, la Tabla 5 y la Tabla 6.

En la Tabla 7 se muestra el cálculo del coste total del proyecto que, como se puede ver, asciende a 24.468,29 €. Como se puede observar, la mayor parte del presupuesto (80%) va destinado a horas internas, ya que el proyecto no requiere demasiados recursos materiales.

Tabla 4. Partida de horas internas del presupuesto del proyecto.

HORAS INTERNAS			
Concepto	**Coste unitario (€/h)**	**Nº unidades (h)**	**Total**
Ingeniero de telecomunicaciones	30	604	18.120,00 €
Director del proyecto	50	30	1.500,00 €
		SUBTOTAL	19.620,00 €

Tabla 5. Partida de amortizaciones del presupuesto del proyecto.

AMORTIZACIONES				
Concepto	**Coste de adquisición**	**Vida útil (años)**	**Uso (meses)**	**TOTAL**
Ordenador portátil ASUS ZenBook	1.199,00 €	5	8	159,87 €
Raspberry Pi 3	60,00 €	5	4	4,00 €
Servidor Supermicro	2.000 €	8	4	83,33 €
Licencia Windows 10 Pro	259,00 €	5	8	34,53 €
Licencia Microsoft 365	69,00 €	1	8	46,00 €
			SUBTOTAL	327,73 €

Tabla 6. Partida de gastos del presupuesto del proyecto.

GASTOS			
Concepto	**Coste unitario**	**Nº unidades**	**Total**
Gasto eléctrico	0,117 €/kWh	2000 kWh	234,00 €
Material de oficina	-	-	40,00 €
		SUBTOTAL	274,00 €

Tabla 7. Cálculo del coste total del proyecto.

COSTES DIRECTOS	Horas internas	19.620,00 €	
	Amortizaciones	327,73 €	20.221,73 €
	Gastos	274,00 €	
COSTES INDIRECTOS	10% de costes directos		2.022,17 €
		SUBTOTAL 1	22.243,90 €
IMPREVISTOS	10% del subtotal 1		2.224,39 €
		TOTAL (subtotal 1 + imprevistos)	24.468,29 €

11. Conclusiones

El objetivo principal de este proyecto era diseñar y desplegar una arquitectura de microservicios específica para un entorno de *edge computing* de industria 4.0, con la ayuda de Kubernetes. Tras analizar las necesidades de los entornos de *edge computing*, así como las de la industria 4.0, se han analizado las distribuciones de K8s existentes actualmente y distribuidas para el *edge*. Para ello, se ha propuesto una metodología de medida del rendimiento y el uso de recursos de cada distribución. De entre todas las distribuciones analizadas se ha seleccionado KubeEdge como la alternativa más completa.

KubeEdge es la distribución específica para el *edge* más madura y con mayor crecimiento a futuro, ha registrado los mejores resultados en las pruebas de rendimiento, y ofrece múltiples características y servicios muy útiles para entornos *edge*, como son los gemelos digitales y la autonomía de los nodos *edge*. La funcionalidad de gemelo digital resulta muy útil para monitorizar dispositivos de forma remota, pero la autonomía de los nodos *edge* es una característica esencial para los entornos de *edge computing*. Estas dos características no las soporta K8s de forma operativa, por lo que resulta imprescindible utilizar una distribución como KubeEdge.

A lo largo del trabajo se ha demostrado cómo KubeEdge permite almacenar el estado de un dispositivo como instancia en el *edge* y transmitir dicho estado al *cloud*, manteniendo ambas copias sincronizadas cuando existe conectividad entre ambos. Cuando no existe conectividad, los servicios del *edge* siguen funcionando sin que los contenedores se trasladen desde el controlador o nodo *master* a otro nodo que sí está disponible y con conectividad, a diferencia de lo que ocurriría si se hubiese utilizado únicamente un despliegue *vanilla* K8s.

Después de implementar una arquitectura de monitorización para dispositivos de una planta industrial como prueba de concepto, se puede concluir que KubeEdge es la distribución más apropiada entre las existentes actualmente para un despliegue de microservicios en el *edge*. De esta forma, se ha conseguido desplegar una aplicación que aporta nuevos servicios de gestión de planta a las empresas de la industria 4.0, para que estas puedan tomar decisiones y gestionar sus dispositivos de forma remota y más eficiente.

12. Referencias

[1] *Basque Digital Innovation Hub*. [En línea] Disponible en: https://basqueindustry.spri.eus/es/

[2] CNCF (2021). *Kubernetes at the edge survey report*. [En línea] Disponible en: https://www.cncf.io/wp-content/uploads/2021/05/KubernetesEdge_Survey_Report_2021_v2.pdf

[3] Grand View Research (2021). *Edge Computing Market Size, Share & Trends Analysis Report*. [En línea] Disponible en: https://www.grandviewresearch.com/industry-analysis/edge-computing-market

[4] StrategyR (2020). *Edge Computing Global Market Trajectory & Analytics*. [En línea] Disponible en: https://www.strategyr.com/market-report-edge-computing-forecasts-global-industry-analysts-inc.asp

[5] T. Zonta, C.A. da Costa, R. da Rosa Righi, M.J. de Lima, E. Silveira da Trindade, G. Pyng Li, "*Predictive maintenance in the Industry 4.0: A systematic literature review*", Computers & Industrial Engineering, vol. 150, 2020. [En línea] Disponible en: https://doi.org/10.1016/j.cie.2020.106889

[6] Fiona Treacy (2020). *Making the Move to Industry 4.0*. [En línea] Disponible en: https://www.machinedesign.com/automation-iiot/article/21128329/making-the-move-to-industry-40

[7] Force Technology. *Digital twins*. [En línea] Disponible en: https://forcetechnology.com/en/services/digital-twins

[8] Euskaltel (2020). *El 5G en la industria del futuro*. [En línea] Disponible en: https://blog.euskaltel.com/empresas/el-5g-en-la-industria-del-futuro/

[9] Gestamp (2020). *Telefónica and Gestamp promote the digitalization of the industry with a 5G-connected factory case*. [En línea] Disponible en: https://www.gestamp.com/Media/Press/Press-Releases/2020/Telefonica-and-Gestamp-promote-the-digitalization-of-the-industry-with-a-5G-connected-factory-case

[10] *MQTT Documentation*. [En línea] Disponible en: https://mqtt.org/

[11] Christopher Tozzi (2021). *Why 'Edge Computing vs. Cloud Computing' Misses the Point*. [En línea] Disponible en: https://www.itprotoday.com/hybrid-cloud/why-edge-computing-vs-cloud-computing-misses-point

[12] Docker. [En línea] Disponible en: https://www.docker.com/

[13] containerd. [En línea] Disponible en: https://containerd.io/

[14] CRI-O. [En línea] Disponible en: https://cri-o.io/

[15] Linux Containers. [En línea] Disponible en: https://linuxcontainers.org/

[16] Kubernetes. *Kubernetes Documentation - What is Kubernetes?* [En línea] Disponible en: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[17] CNCF. *Graduated and Incubating Projects*. [En línea] Disponible en: https://www.cncf.io/projects/

[18] Ali Kahoot (2019). *K8s: Deployments vs StatefulSets vs DaemonSets*. [En línea] Disponible en: https://medium.com/stakater/k8s-deployments-vs-statefulsets-vs-daemonsets-60582f0c62d4

[19] Kubernetes. *Kubernetes Documentation - Components*. [En línea] Disponible en: https://kubernetes.io/docs/concepts/overview/components/

[20] CNCF (2019). *Kubernetes Project Journey Report*. [En línea] Disponible en: https://www.cncf.io/reports/cncf-kubernetes-project-journey/

[21] KubeEdge. *KubeEdge Documentation*. https://kubeedge.io/en/docs/

[22] *OpenYurt Github*. [En línea] Disponible en: https://github.com/openyurtio/openyurt

[23] Gokul Chandra (2021). *SuperEdge, OpenYurt - Extending Native Kubernetes to Edge*. [En línea] Disponible en: https://itnext.io/superedge-openyurt-extending-native-kubernetes-to-edge-cc59094f92c

[24] Alibaba Developer (2021). *OpenYurt v0.3.0 Released: Improve Application Deployment Efficiency in Edge Scenarios*. [En línea] Disponible en: https://www.alibabacloud.com/blog/openyurt-v0-3-0-released-improve-application-deployment-efficiency-in-edge-scenarios_597183

[25] k3s. *k3s Documentation*. [En línea] Disponible en: https://rancher.com/docs/k3s/latest/en/

[26] Hrittik Roy (2021). *K8s vs k3s: The Comprehensive Difference*. [En línea] Disponible en: https://www.p3r.one/k8s-vs-k3s/

[27] MicroK8s. *MicroK8s Documentation*. [En línea] Disponible en: https://microk8s.io/docs

[28] Kubermatic. *Kubermatic Documentation*. [En línea] Disponible en: https://docs.kubermatic.com/kubermatic

[29] KubeOne. *KubeOne Documentation*. [En línea] Disponible en: https://docs.kubermatic.com/kubeone

[30] *k0s Github*. [En línea] Disponible en: https://github.com/k0sproject/k0s

[31] *SuperEdge Github*. [En línea] Disponible en: https://github.com/superedge/superedge

[32] *Determining the number of iterations*. The Gnumeric Manual, version 1.10. [En línea] Disponible en: http://www.hep.by/gnu/gnumeric/sect-advanced-analysis-simulation-iterations.shtml

[33] Manual del comando ps. [En línea] Disponible en: https://man7.org/linux/man-pages/man1/ps.1.html

[34] *Sysbench Documentation*. [En línea] Disponible en: https://wiki.gentoo.org/wiki/Sysbench

[35] *Flourish*. Disponible en: https://flourish.studio/

[36] *Mosquitto Documentation* [En línea] Disponible en: https://mosquitto.org/

[37] *KubeEdge Counter Demo*. [En línea] Disponible en: https://github.com/kubeedge/examples/blob/master/kubeedge-counter-demo/README.md

[38] *Grafana Documentation* [En línea] Disponible en: https://grafana.com/docs/grafana/latest/getting-started/

13. Anexo I

13.1. Plan de pruebas

13.1.1. Pruebas de rendimiento

A continuación se muestra el contenido del *script*, desarrollado en lenguaje bash, ejecutado para obtener los resultados de las pruebas de rendimiento de CPU, memoria y disco. Como ya se menciona en el apartado 6.3, el *script* se basa en el uso de las herramientas Sysbench y FIO. Ejecuta las pruebas en 30 iteraciones y almacena los resultados de cada tipo de prueba en un fichero CSV separado.

```
#!/bin/bash

TEST=""
SUPPORTED_TESTS=("cpu" "mem" "io" "all")

NUM_REPETITIONS=30

# The size of the test file should be larger than the RAM memory to ensure
# that the file cache will not affect the test too much;
IO_FILE_SIZE=12G

CURRENT_TIME=$(date +"%s")

BASE_DIR="."
CPU_FILE="$BASE_DIR/cpu_test_results-${CURRENT_TIME}.csv"
MEM_W_FILE="$BASE_DIR/mem_w_test_results-${CURRENT_TIME}.csv"
MEM_R_FILE="$BASE_DIR/mem_r_test_results-${CURRENT_TIME}.csv"
IO_FILE="$BASE_DIR/io_test_results-${CURRENT_TIME}.csv"

function usage(){
    echo -e "usage: $0 [OPTIONS]"
    echo -e "Perform CPU, memory or disk I/O test benchmarks"
    echo -e "  OPTIONS:"
    echo -e "    -t <cpu/mem/io/all>:   Test benchmark to perform (all by default)"
    echo -e "    -h / --help:           Print this help"
}

while getopts ":h-:t:" o; do
    case "${o}" in
        h)
            usage && exit 0
            ;;
        t)
            TEST="${OPTARG}"
            if [[ ! "${SUPPORTED_TESTS[@]}" =~ "${TEST}" ]]; then
              echo -e "Test '${TEST}' not supported\n" >&2
              echo -e "Supported options are: [${SUPPORTED_TESTS[*]}]\n" >&2
              usage && exit 1
            elif [[ "${TEST}" == "all" ]]; then
              PERFORM_TESTS=("cpu" "mem" "io")
            else
              PERFORM_TESTS=("${TEST}")
            fi
```

```
            ;;
        -)
            [ "${OPTARG}" == "help" ] && usage && exit 0
            echo -e "Invalid option: '--${OPTARG}'\n" >&2
            usage && exit 1
            ;;
        \?)
            echo -e "Invalid option: '-${OPTARG}'\n" >&2
            usage && exit 1
            ;;
        *)
            usage && exit 1
            ;;
    esac
done

function cpu_test() {
  CPU_OUTPUT=$(sysbench cpu run)
  EVENTS_PER_SECOND=$(echo "${CPU_OUTPUT}" | grep "events per second" | awk '{print
$4}')

  echo -e "\tEvents per second: ${EVENTS_PER_SECOND}\n"

  cpu_results+=(${EVENTS_PER_SECOND})
  echo "${EVENTS_PER_SECOND}" >> "${CPU_FILE}"
}

function mem_test_write() {
  MEM_OUTPUT=$(sysbench memory run --memory-oper=write)
  OPERATIONS_PER_SECOND=$(echo "${MEM_OUTPUT}" | grep "Total operations" | awk
'{print $4}' | cut -c 2-)
  THROUGHPUT_MBPS=$(echo "${MEM_OUTPUT}" | grep "Total operations" -A 3 | sed -n '3
p' | awk '{print $4}' | cut -c 2-)

  echo -e "\tOperations per second: ${OPERATIONS_PER_SECOND}"
  echo -e "\tThroughput: ${THROUGHPUT_MBPS} Mbps\n"

  mem_w_operations_results+=(${OPERATIONS_PER_SECOND})
  mem_w_throughput_results+=(${THROUGHPUT_MBPS})
  echo -e "${OPERATIONS_PER_SECOND}\t${THROUGHPUT_MBPS}" >> "${MEM_W_FILE}"
}

function mem_test_read() {
  MEM_OUTPUT=$(sysbench memory run --memory-oper=read)
  OPERATIONS_PER_SECOND=$(echo "${MEM_OUTPUT}" | grep "Total operations" | awk
'{print $4}' | cut -c 2-)
  THROUGHPUT=$(echo "${MEM_OUTPUT}" | grep "Total operations" -A 3 | sed -n '3 p' |
awk '{print $4}' | cut -c 2-)

  echo -e "\tOperations per second: ${OPERATIONS_PER_SECOND}"
  echo -e "\tThroughput: ${THROUGHPUT} Mbps\n"

  mem_r_operations_results+=(${OPERATIONS_PER_SECOND})
  mem_r_throughput_results+=(${THROUGHPUT_MBPS})
  echo -e "${OPERATIONS_PER_SECOND}\t${THROUGHPUT_MBPS}" >> "${MEM_R_FILE}"
}

function io_test() {
```

```
  IO_OUTPUT=$(sysbench fileio --file-total-size=$IO_FILE_SIZE --file-test-mode=rndrw
--max-time=60 --max-requests=0 --file-extra-flags=direct run)

  IOPS_W=$(echo "${IO_OUTPUT}" | grep "File operations:" -A 3 | sed -n '3 p' | awk
'{print $2}')
  IOPS_R=$(echo "${IO_OUTPUT}" | grep "File operations:" -A 3 | sed -n '2 p' | awk
'{print $2}')
  THROUGHPUT_W=$(echo "${IO_OUTPUT}" | grep "Throughput:" -A 3 | sed -n '3 p' | awk
'{print $3}')
  THROUGHPUT_R=$(echo "${IO_OUTPUT}" | grep "Throughput:" -A 3 | sed -n '2 p' | awk
'{print $3}')

  echo -e "\tWrite operations per second: ${IOPS_W}"
  echo -e "\tWrite throughput: ${THROUGHPUT_W} Mbps"
  echo -e "\tRead operations per second: ${IOPS_R}"
  echo -e "\tRead throughput: ${THROUGHPUT_R} Mbps\n"

  io_w_operations_results+=(${IOPS_W})
  io_w_throughput_results+=(${THROUGHPUT_W})
  io_r_operations_results+=(${IOPS_R})
  io_r_throughput_results+=(${THROUGHPUT_R})
  echo -e "${IOPS_W}\t${THROUGHPUT_W}\t${IOPS_R}\t${THROUGHPUT_R}" >> "${IO_FILE}"
}

function mean_value() {
  FILENAME=$1
  VALUE_NAME=$2
  shift 2
  VALUES=("$@")

  SUM=$(IFS="+"; bc <<< "${VALUES[*]}")
  AVERAGE=$(echo $SUM / ${#VALUES[@]} | bc -l)

  echo "Average ${VALUE_NAME}: ${AVERAGE}"
  echo "Average ${VALUE_NAME}: ${AVERAGE}" >> "${FILENAME}"
}

if [ -z "${TEST}" ]; then
  TEST="all"
  PERFORM_TESTS=("cpu" "mem" "io")
fi

echo -e "Performing following tests: ${PERFORM_TESTS[*]}"

for test in "${PERFORM_TESTS[@]}"
do
  case "${test}" in
    "cpu")
        echo -e "\n--------------------------------------\n"

        cpu_results=()
        for i in $(seq $NUM_REPETITIONS); do
          echo -e "Performing CPU test, iteration $i"
          cpu_test
        done
        mean_value $CPU_FILE "events per second" "${cpu_results[@]}"
        ;;

    "mem")
```

```
        echo -e "\n--------------------------------------\n"

        mem_w_operations_results=()
        mem_w_throughput_results=()
        for i in $(seq $NUM_REPETITIONS); do
          echo -e "Performing memory write test, iteration $i"
          mem_test_write
        done
        mean_value $MEM_W_FILE "write operations per second"
"${mem_w_operations_results[@]}"
        mean_value $MEM_W_FILE "write throughput" "${mem_w_throughput_results[@]}"

        echo -e "\n--------------------------------------\n"

        mem_r_operations_results=()
        mem_r_throughput_results=()
        for i in $(seq $NUM_REPETITIONS); do
          echo -e "Performing memory read test, iteration $i"
          mem_test_read
        done
        mean_value $MEM_R_FILE "read operations per second"
"${mem_r_operations_results[@]}"
        mean_value $MEM_R_FILE "read throughput" "${mem_r_throughput_results[@]}"
        ;;

    "io")
        echo -e "\n--------------------------------------\n"

        echo -e "Creating files (total size: $IO_FILE_SIZE)...\n"
        sysbench fileio --file-total-size=$IO_FILE_SIZE prepare > /dev/null 2>&1

        io_w_operations_results=()
        io_w_throughput_results=()
        io_r_operations_results=()
        io_r_throughput_results=()
        for i in $(seq $NUM_REPETITIONS); do
          echo -e "Performing disk I/O test, iteration $i"
          io_test
        done

        echo -e "Cleaning files...\n"
        sysbench fileio --file-total-size=$IO_FILE_SIZE cleanup > /dev/null 2>&1

        mean_value $IO_FILE "write operations per second"
"${io_w_operations_results[@]}"
        mean_value $IO_FILE "write throughput" "${io_w_throughput_results[@]}"
        mean_value $IO_FILE "read operations per second"
"${io_r_operations_results[@]}"
        mean_value $IO_FILE "read throughput" "${io_r_throughput_results[@]}"
        ;;

    *)
        echo -e "[ERROR] Test '${test}' not valid\n"
        exit 1
        ;;
  esac
done
```

Figura 41. *Script* de ejecución de pruebas de rendimiento.

14. Anexo II

14.1. Código

En este apartado se incluyen los ficheros de configuración de las aplicaciones y de los despliegues de cada uno de los microservicios descritos en el apartado 7.1.2.

14.1.1. Microservicio de almacenamiento de datos y estadísticas

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

Figura 42. Fichero YAML con descripción del almacenamiento persistente.

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
```

```
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
          - name: MYSQL_ROOT_PASSWORD
            value: "password"
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-pers-storage
          mountPath: /var/lib/mysql
        - name: mysql-initdb
          mountPath: /docker-entrypoint-initdb.d
      volumes:
      - name: mysql-pers-storage
        persistentVolumeClaim:
          claimName: mysql-pvc
      - name: mysql-initdb
        configMap:
          name: mysql-initdb-config
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-initdb-config
data:
  initdb.sql: |
    CREATE DATABASE tfm;
    USE tfm;
    CREATE TABLE sensor_data (
        id INT NOT NULL AUTO_INCREMENT,
        time DATETIME NOT NULL,
        signal_sin1 FLOAT(24) NOT NULL,
        signal_sin2 FLOAT(24) NOT NULL,
        signal_triang FLOAT(24) NOT NULL,
        signal_linear FLOAT(24) NOT NULL,
        signal_random FLOAT(24) NOT NULL,
        signal_constant FLOAT(24) NOT NULL,
        PRIMARY KEY (id)
    );
    CREATE TABLE sensor_data_stats_max (
        id INT NOT NULL AUTO_INCREMENT,
        time DATETIME NOT NULL,
        signal_sin1 FLOAT(24) NOT NULL,
        signal_sin2 FLOAT(24) NOT NULL,
        signal_triang FLOAT(24) NOT NULL,
        signal_linear FLOAT(24) NOT NULL,
        signal_random FLOAT(24) NOT NULL,
        signal_constant FLOAT(24) NOT NULL,
        PRIMARY KEY (id)
```

```
    );
    CREATE TABLE sensor_data_stats_min (
        id INT NOT NULL AUTO_INCREMENT,
        time DATETIME NOT NULL,
        signal_sin1 FLOAT(24) NOT NULL,
        signal_sin2 FLOAT(24) NOT NULL,
        signal_triang FLOAT(24) NOT NULL,
        signal_linear FLOAT(24) NOT NULL,
        signal_random FLOAT(24) NOT NULL,
        signal_constant FLOAT(24) NOT NULL,
        PRIMARY KEY (id)
    );
    CREATE TABLE sensor_data_stats_mean (
        id INT NOT NULL AUTO_INCREMENT,
        time DATETIME NOT NULL,
        signal_sin1 FLOAT(24) NOT NULL,
        signal_sin2 FLOAT(24) NOT NULL,
        signal_triang FLOAT(24) NOT NULL,
        signal_linear FLOAT(24) NOT NULL,
        signal_random FLOAT(24) NOT NULL,
        signal_constant FLOAT(24) NOT NULL,
        PRIMARY KEY (id)
    );
    CREATE TABLE sensor_data_stats_stdev (
        id INT NOT NULL AUTO_INCREMENT,
        time DATETIME NOT NULL,
        signal_sin1 FLOAT(24) NOT NULL,
        signal_sin2 FLOAT(24) NOT NULL,
        signal_triang FLOAT(24) NOT NULL,
        signal_linear FLOAT(24) NOT NULL,
        signal_random FLOAT(24) NOT NULL,
        signal_constant FLOAT(24) NOT NULL,
        PRIMARY KEY (id)
    );
```

Figura 43. Fichero YAML con descripción del despliegue y el servicio *mysql*.

14.1.2. Microservicio de generación de datos

```
import os

GEN_INTERVAL = 1    # seconds
NUM_PERIODS = 2

SIN_PERIOD_1 = 8    # seconds
SIN_PERIOD_2 = 24   # seconds
TRIANG_PERIOD = 6   # seconds

MQTT_TOPIC = "mocksensor"
MQTT_BROKER_HOST = os.environ['MQTT_BROKER_HOST']
```

Figura 44. *Script* "constants.py" con la definición de constantes de la aplicación.

```
import json
import random
import time
from datetime import datetime

import numpy as np
```

```
import paho.mqtt.publish as publish
from scipy import signal as sg

import constants as c


def sin_wave_generator(frequency):
    chunk_length = int(c.NUM_PERIODS / frequency)
    sample_rate = chunk_length
    t = np.linspace(0, chunk_length, sample_rate, False)
    signal = np.sin(2 * np.pi * frequency * t)
    return signal


def triangular_wave_generator(frequency):
    chunk_length = int(c.NUM_PERIODS / frequency)
    sample_rate = chunk_length
    t = np.linspace(0, chunk_length, sample_rate, False)
    signal = sg.sawtooth(2 * np.pi * frequency * t, width=0.5)
    return signal


def mqtt_publish_dict(dict_data):
    publish.single(c.MQTT_TOPIC, json.dumps(dict_data), hostname=c.MQTT_BROKER_HOST)
    print("Sent data to topic %s: %s" % (c.MQTT_TOPIC, dict_data))


if __name__ == "__main__":
    counter = 0
    system_random = random.SystemRandom()
    signal_sin1 = []
    signal_sin1_index = 0
    signal_sin2 = []
    signal_sin2_index = 0
    signal_triang = []
    signal_triang_index = 0

    while True:
        if signal_sin1_index == len(signal_sin1):
            print("Generating signal_sin1...")
            signal_sin1 = sin_wave_generator(1 / c.SIN_PERIOD_1)
            signal_sin1_index = 0
            print("Length: %s" % len(signal_sin1))
        if signal_sin2_index == len(signal_sin2):
            print("Generating signal_sin2...")
            signal_sin2 = sin_wave_generator(1 / c.SIN_PERIOD_2)
            signal_sin2_index = 0
            print("Length: %s" % len(signal_sin2))
        if signal_triang_index == len(signal_triang):
            print("Generating signal_triang...")
            signal_triang = triangular_wave_generator(1 / c.TRIANG_PERIOD)
            signal_triang_index = 0
            print("Length: %s" % len(signal_triang))

        signal_linear = counter / 1000
        signal_random = system_random.random()
        signal_constant = 10

        data_dict = {
```

```
            "time": datetime.now().strftime('%Y-%m-%d %H:%M:%S'),
            "signal_sin1": signal_sin1[signal_sin1_index],
            "signal_sin2": signal_sin2[signal_sin2_index],
            "signal_triang": signal_triang[signal_triang_index],
            "signal_linear": signal_linear,
            "signal_random": signal_random,
            "signal_constant": signal_constant
        }
        mqtt_publish_dict(data_dict)

        time.sleep(1)
        counter += 1
        signal_sin1_index += 1
        signal_sin2_index += 1
        signal_triang_index += 1
```

Figura 45. *Script* "generator.py" con la generación de datos y envío por MQTT.

```
FROM python:3

WORKDIR /usr/src/tfm

COPY . .
RUN pip install --no-cache-dir -r requirements.txt

CMD [ "python", "./generator.py" ]
```

Figura 46. *Dockerfile* para generar la imagen del generador de datos.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: data-generation
  labels:
    app: data-generation
spec:
  replicas: 1
  selector:
    matchLabels:
      app: data-generation
  template:
    metadata:
      labels:
        app: data-generation
    spec:
      containers:
      - name: data-generation
        image: registry.i2t.local/datageneration:latest
        env:
          - name: MQTT_BROKER_HOST
            valueFrom:
              fieldRef:
                fieldPath: status.hostIP
```

Figura 47. Fichero YAML con descripción del despliegue *data-generation*.

14.1.3. Microservicio de lectura de datos MQTT

```
import os

MQTT_TOPIC = "mocksensor"
MQTT_BROKER_HOST = os.environ['MQTT_BROKER_HOST']

MYSQL_SERVER_HOST = os.environ['MYSQL_SERVER_HOST']
MYSQL_USER = os.environ['MYSQL_USER']
MYSQL_PASS = os.environ['MYSQL_PASS']

MYSQL_DB = "tfm"
MYSQL_SENSOR_DATA_TABLE = "sensor_data"

SQL_INSERT_SENSOR_DATA = "INSERT INTO " + \
                         MYSQL_SENSOR_DATA_TABLE + \
                         " (time, signal_sin1, signal_sin2," \
                         " signal_triang, signal_linear, signal_random," \
                         " signal_constant) VALUES (%s, %s, %s, %s, %s, %s, %s)"

NUM_RECORDS_TO_KEEP = 100
SQL_KEEP_LAST_N_RECORDS = "DELETE FROM " + MYSQL_SENSOR_DATA_TABLE + \
                          " WHERE id NOT IN (SELECT id FROM (SELECT id FROM " + \
                          MYSQL_SENSOR_DATA_TABLE + " ORDER BY id " \
                          "DESC LIMIT " + str(NUM_RECORDS_TO_KEEP) + ") foo);"
```

Figura 48. *Script* "constants.py" con la definición de constantes de la aplicación.

```
import json

import mysql.connector
import paho.mqtt.subscribe as subscribe

import constants as c


def read_data():
    mqtt_msg = subscribe.simple(c.MQTT_TOPIC, hostname=c.MQTT_BROKER_HOST)
    data = json.loads(mqtt_msg.payload.decode("utf-8"))
    print("-----------------------------------")
    print("TIME: %s" % data["time"])
    print("SIN1: %s" % data["signal_sin1"])
    print("SIN2: %s" % data["signal_sin2"])
    print("TRIANG: %s" % data["signal_triang"])
    print("LINEAR: %s" % data["signal_linear"])
    print("RANDOM: %s" % data["signal_random"])
    print("CONSTANT: %s" % data["signal_constant"])
    print("-----------------------------------")
    return data


if __name__ == '__main__':
    mydb = mysql.connector.connect(
        host=c.MYSQL_SERVER_HOST,
        user=c.MYSQL_USER,
        password=c.MYSQL_PASS,
        database=c.MYSQL_DB
    )
    mycursor = mydb.cursor()
```

```
    while True:
        data = read_data()
        sql_data = (data["time"], data["signal_sin1"],
                    data["signal_sin2"], data["signal_triang"],
                    data["signal_linear"], data["signal_random"],
                    data["signal_constant"])
        mycursor.execute(c.SQL_INSERT_SENSOR_DATA, sql_data)
        mycursor.execute(c.SQL_KEEP_LAST_N_RECORDS)
        mydb.commit()
```

Figura 49. *Script* "mqtt-subscriber.py" con la lectura MQTT y escritura en la BD.

```
FROM python:3

WORKDIR /usr/src/tfm

COPY . .
RUN pip install --no-cache-dir -r requirements.txt

CMD [ "python", "./mqtt-subscriber.py" ]
```

Figura 50. *Dockerfile* para generar la imagen de la lectura de datos MQTT.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: data-subscriber
  labels:
    app: data-subscriber
spec:
  replicas: 1
  selector:
    matchLabels:
      app: data-subscriber
  template:
    metadata:
      labels:
        app: data-subscriber
    spec:
      containers:
      - name: data-subscriber
        image: registry.i2t.local/datasubscriber:latest
        env:
          - name: MYSQL_USER
            value: "root"
          - name: MYSQL_PASS
            value: "password"
          - name: MQTT_BROKER_HOST
            valueFrom:
              fieldRef:
                fieldPath: status.hostIP
```

Figura 51. Fichero YAML con descripción del despliegue *data-subscriber*.

14.1.4. Microservicio de cálculo de estadísticas

```
import os

MYSQL_SERVER_HOST = os.environ['MYSQL_SERVER_HOST']
MYSQL_USER = os.environ['MYSQL_USER']
MYSQL_PASS = os.environ['MYSQL_PASS']

MYSQL_DB = "tfm"
MYSQL_SENSOR_DATA_TABLE = "sensor_data"
MYSQL_SENSOR_DATA_STATS_TABLE = "sensor_data_stats_"

SQL_SELECT_SENSOR_DATA = "SELECT * FROM " + MYSQL_SENSOR_DATA_TABLE
SQL_INSERT_SENSOR_DATA_STATS = "INSERT INTO " + MYSQL_SENSOR_DATA_STATS_TABLE + \
                               "%s (time, signal_sin1," \
                               "signal_sin2, signal_triang, signal_linear," \
                               "signal_random, signal_constant) VALUES (%%s, %%s,
%%s, %%s, %%s, %%s, %%s)"

NUM_RECORDS_TO_KEEP = 10
SQL_KEEP_LAST_N_RECORDS = "DELETE FROM " + MYSQL_SENSOR_DATA_STATS_TABLE + \
                          "%s WHERE id NOT IN (SELECT id FROM (SELECT id FROM " + \
                          MYSQL_SENSOR_DATA_STATS_TABLE + "%s ORDER BY id " \
                          "DESC LIMIT " + str(NUM_RECORDS_TO_KEEP) + ") foo);"
```

Figura 52. *Script* "constants.py" con la definición de constantes de la aplicación.

```
import statistics
import time
from datetime import datetime

import mysql.connector

import constants as c


def calculate_max(value_list):
    return max(value_list)


def calculate_min(value_list):
    return min(value_list)


def calculate_mean(value_list):
    return statistics.mean(value_list)


def calculate_stdev(value_list):
    return statistics.stdev(value_list)


if __name__ == '__main__':

    while True:
        mydb = mysql.connector.connect(
            host=c.MYSQL_SERVER_HOST,
            user=c.MYSQL_USER,
            password=c.MYSQL_PASS,
```

```
            database=c.MYSQL_DB
        )

        mycursor = mydb.cursor()

        signal_sin1 = []
        signal_sin2 = []
        signal_triang = []
        signal_linear = []
        signal_random = []
        signal_constant = []

        mycursor.execute(c.SQL_SELECT_SENSOR_DATA)
        data_results = mycursor.fetchall()
        for row in data_results:
            signal_sin1.append(row[2])
            signal_sin2.append(row[3])
            signal_triang.append(row[4])
            signal_linear.append(row[5])
            signal_random.append(row[6])
            signal_constant.append(row[7])

        current_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

        print(current_time)
        print("[SIGNAL_SIN1] max: %s; min:%s mean: %s; stdev: %s" %
              (calculate_max(signal_sin1), calculate_min(signal_sin1),
               calculate_mean(signal_sin1), calculate_stdev(signal_sin1)))
        print("[SIGNAL_SIN2] max: %s; min:%s mean: %s; stdev: %s" %
              (calculate_max(signal_sin2), calculate_min(signal_sin2),
               calculate_mean(signal_sin2), calculate_stdev(signal_sin2)))
        print("[SIGNAL_TRIANG] max: %s; min:%s mean: %s; stdev: %s" %
              (calculate_max(signal_triang), calculate_min(signal_triang),
               calculate_mean(signal_triang), calculate_stdev(signal_triang)))
        print("[SIGNAL_LINEAR] max: %s; min:%s mean: %s; stdev: %s" %
              (calculate_max(signal_linear), calculate_min(signal_linear),
               calculate_mean(signal_linear), calculate_stdev(signal_linear)))
        print("[SIGNAL_RANDOM] max: %s; min:%s mean: %s; stdev: %s" %
              (calculate_max(signal_random), calculate_min(signal_random),
               calculate_mean(signal_random), calculate_stdev(signal_random)))
        print("[SIGNAL_CONSTANT] max: %s; min:%s mean: %s; stdev: %s" %
              (calculate_max(signal_constant), calculate_min(signal_constant),
               calculate_mean(signal_constant), calculate_stdev(signal_constant)))

        mycursor.execute(c.SQL_INSERT_SENSOR_DATA_STATS % "max",
                         (current_time, calculate_max(signal_sin1),
calculate_max(signal_sin2),
                          calculate_max(signal_triang),
calculate_max(signal_linear),
                          calculate_max(signal_random),
calculate_max(signal_constant)))
        mycursor.execute(c.SQL_INSERT_SENSOR_DATA_STATS % "min",
                         (current_time, calculate_min(signal_sin1),
calculate_min(signal_sin2),
                          calculate_min(signal_triang),
calculate_min(signal_linear),
                          calculate_min(signal_random),
calculate_min(signal_constant)))
        mycursor.execute(c.SQL_INSERT_SENSOR_DATA_STATS % "mean",
```

```
                        (current_time, calculate_mean(signal_sin1),
calculate_mean(signal_sin2),
                         calculate_mean(signal_triang),
calculate_mean(signal_linear),
                         calculate_mean(signal_random),
calculate_mean(signal_constant)))
        mycursor.execute(c.SQL_INSERT_SENSOR_DATA_STATS % "stdev",
                        (current_time, calculate_stdev(signal_sin1),
calculate_stdev(signal_sin2),
                         calculate_stdev(signal_triang),
calculate_stdev(signal_linear),
                         calculate_stdev(signal_random),
calculate_stdev(signal_constant)))
        mycursor.execute(c.SQL_KEEP_LAST_N_RECORDS % ("max", "max"))
        mycursor.execute(c.SQL_KEEP_LAST_N_RECORDS % ("min", "min"))
        mycursor.execute(c.SQL_KEEP_LAST_N_RECORDS % ("mean", "mean"))
        mycursor.execute(c.SQL_KEEP_LAST_N_RECORDS % ("stdev", "stdev"))
        mydb.commit()
        time.sleep(10)
```

Figura 53. *Script* "statistics-calculator.py" con el cálculo de estadísticas.

```
FROM python:3

WORKDIR /usr/src/tfm

COPY . .
RUN pip install --no-cache-dir -r requirements.txt

CMD [ "python", "./statistics-calculator.py" ]
```

Figura 54. *Dockerfile* para generar la imagen de la calculadora de estadísticas.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: data-processing
  labels:
    app: data-processing
spec:
  replicas: 1
  selector:
    matchLabels:
      app: data-processing
  template:
    metadata:
      labels:
        app: data-processing
    spec:
      containers:
      - name: data-processing
        image: registry.i2t.local/dataprocessing:latest
        env:
          - name: MYSQL_USER
            value: "root"
          - name: MYSQL_PASS
            value: "password"
```

Figura 55. Fichero YAML con descripción del despliegue *data-processing*.