Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Sistema de gestión de software para dispositivos IoT.

Autor: Miguel Ángel Cabrera Miñagorri
Tutor: Antonio Jesus Sierra Collado

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Sistema de gestión de software para dispositivos IoT.

Autor:
Miguel Ángel Cabrera Miñagorri

Tutor:
Antonio Jesus Sierra Collado
Profesor Sustituto Interino

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Trabajo Fin de Grado: Sistema de gestión de software para dispositivos IoT.

Autor: Miguel Ángel Cabrera Miñagorri

Tutor: Antonio Jesus Sierra Collado

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

A mi familia

A mis maestros

Resumen

Uno de los problemas actuales del Internet de las cosas (Internet of Things, IoT) es el despliegue de aplicaciones nuevas o actualización de antiguas sobre los dispositivos de forma remota. Existen problemas debidos a las dependencias, sus versiones y compatibilidad de estas con las nuevas actualizaciones. Otros problemas vienen derivados de que los errores que se producen en una aplicación puedan afectar a otras, o incluso afecten a componentes principales del dispositivo, es decir, problemas de aislamiento, y además puede existir una falta de escalabilidad de la red.

Durante el desarrollo de este proyecto se han utilizado contenedores para ejecutar aplicaciones en los dispositivos IoT y solucionar los problemas de aislamiento y dependencias. Para la ejecución de los contendores se hace uso de dos motores de contendores de código abierto, Docker y balena-engine. Para la gestión de la red de dispositivos se ha usado Kubernetes, un sistema de orquestación de contenedores. Gracias a Kubernetes se ha creado un clúster de tipo maestro-esclavo que permite, a través de una API, gestionar los contendores que se ejecutan en los dispositivos, y, por tanto, las aplicaciones. Del uso de Kubernetes se obtienen otras ventajas, como la posibilidad de exponer servicios directamente desde los dispositivos para ser consultados desde el exterior del clúster. Con Kubernetes se solucionan tanto el problema de la escalabilidad como el despliegue.

En esta memoria se explican conceptos básicos tanto de contenedores como de Kubernetes y se profundiza en su funcionamiento interno, algunas características de seguridad, protocolos utilizados, direccionamiento interno, etc. Al ser tecnologías orientadas a la computación en la nube se han adaptado al IoT, por lo que se explica cómo realizar estas adaptaciones para posteriormente ponerlas en práctica. También, se evalúan las características mínimas a nivel de hardware para que un dispositivo pueda incluirse en el sistema.

Para demostrar que la teoría explicada realmente puede ponerse en práctica, se ha creado un clúster de Kubernetes de pruebas, desde cero. Para ello, se ha hecho uso de Kubeadm y Kubelet, y se ha utilizado Kubectl para dar las órdenes a través de la API que expone el maestro. El clúster de pruebas consta de dos Raspberry Pi, que juegan el papel de dispositivos IoT y son los esclavos, y por una máquina virtual ejecutándose en un ordenador portátil que juega el papel de maestro. El maestro hace uso de Docker para la ejecución de contenedores, mientras que los esclavos hacen uso de balena-engine por motivos de rendimiento y capacidad de computación. Este clúster es híbrido, ya que conviven en él dos motores de contenedores distintos y es multiarquitectura, puesto que el maestro es amd64 y los esclavos arm. Se explica además, cómo deben ser configurados los componentes principales del clúster como Kubelet o el componente encargado del direccionamiento interno, llamado Flannel.

Dado que los dispositivos IoT por lo general utilizan sensores u otro tipo de hardware, se debe garantizar que es posible acceder a estos desde las aplicaciones que se ejecutan en un contenedor. Por ello, se explica el funcionamiento del hardware de una RaspberryPi a través de pines GPIO, cómo la aplicación puede acceder a él desde el contenedor y cómo se deben realizar los despliegues en el clúster para que los contenedores que se lanzan automáticamente puedan hacer uso de los sensores.

Dentro de un contenedor se ejecuta una imagen que contiene la aplicación. Se han creado dos imágenes para ser ejecutadas en el clúster de pruebas. Estas dos imágenes hacen uso de sensores. La primera usa el sensor de luz para enviar a sus logs el nivel de luz externo y la segunda hace uso de diodos led para simular un semáforo. La creación de estas imágenes se utiliza para explicar el proceso de creación y optimización de una imagen cualquiera hasta conseguir que ocupe el mínimo espacio posible, ya que los dispositivos suelen tener poca memoria, y también para que puedan ser descargadas rápidamente desde el registro de imágenes. Como registro de imágenes se ha utilizado Dockerhub.

Por último, se muestran evidencias de la creación del clúster, de la ejecución de las aplicaciones creadas en el clúster de pruebas y se evalúa el grado de obtención de los objetivos fijados.

Abstract

One of the current problems of the Internet of Things (Internet of Things, IoT) is the deployment of new applications or updates of old ones over devices remotely. There are problems due to the dependencies, their versions and their compatibility with the new updates. Other problems arise from the fact that errors that occur in one application may affect others, or even affect the main components of the device, that is, isolation problems, and, also, it could exist a lack of scalability of the network.

During the development of this project containers have been used to run applications on IoT devices and solve the problems of isolation and dependencies. For the execution of the containers, two open source container engines, Docker and balena-engine, are used. For the management of the device network, Kubernetes, a container orchestration system, has been used. Thanks to Kubernetes, a cluster of master-slave type has been created that allows, through an API, to manage the containers that are executed in the devices, and, therefore, the applications. Other advantages are obtained from the use of Kubernetes, such as the possibility of exposing services directly from the devices to be consulted from outside the cluster. With Kubernetes both problems, scalability and deployment have been resolved.

In this report, basic concepts of containers and Kubernetes are explained and their internal functioning, some security features, used protocols, internal addressing, etc. are studied in depth. Being technologies oriented to cloud computing have been adapted to the IoT, so it is explained how to make these adaptations to put them later into practice. Also, the minimum characteristics at the hardware level are evaluated so that a device can be included in the system.

To demonstrate that the explained theory can actually be put into practice, a Kubernetes cluster for tests has been created, from scratch. For this, Kubeadm and Kubelet have been used, and Kubectl has been used to give the orders through the API that the master exposes. The test cluster consists of two Raspberry Pi, which play the role of IoT devices, so they are slaves, and by a virtual machine running on a laptop that plays the role of master. The master makes use of Docker for the execution of containers, while the slaves make use of balena-engine for performance reasons and computing capacity. This cluster is hybrid, since two different container engines coexist in it and it is multi-architecture, since the master is amd64 and the slaves are arm. It also explains how the main components of the cluster should be configured, such as Kubelet or the component responsible for internal addressing, called Flannel.

Since IoT devices usually use sensors or other hardware, it must be guaranteed that they can be accessed from applications running in a container. Therefore, it explains the operation of the hardware of a RaspberryPi through GPIO pins, how the application can access it from the container and how the deployments in the cluster should be carried out so that containers that are launched automatically can make use of the sensors.

An image containing the application is executed inside a container. Two images have been created to be executed in the test cluster. These two images make use of sensors. The first uses the light sensor to send the external light level to its logs and the second uses LED diodes to simulate a traffic light. The creation of these images is used to explain the process of creating and optimizing any image until it takes the minimum space as possible, since the devices usually have little memory, and also so that they can be downloaded quickly from the image registry. Dockerhub has been used as image registry.

Finally, there is evidence of the creation, of the execution of the created applications in the test cluster and the grade of objetives achievement is evaluated.

Índice

ÍNDICE DE FIGURAS

ÍNDICE DE CUADROS DE TEXTO

1 INTRODUCCIÓN, ACTUALIDAD Y OBJETIVOS

Las cosas no se hacen siguiendo caminos distintos para que no sean iguales, sino para que sean mejores.

- Elon Musk -

Uno de los problemas actuales del Internet de las cosas (Internet of Things) en adelante IoT, es el despliegue de aplicaciones nuevas o actualización de antiguas sobre los dispositivos de forma remota. Principalmente se pueden encontrar problemas debidos a las dependencias, sus versiones y compatibilidad de estas con las nuevas actualizaciones. Por ejemplo, si se quisiera desplegar sobre ciertos dispositivos una nueva versión de una aplicación. Tras realizar las modificaciones, y probarla en un entorno de pruebas, se decidió que era hora de trasladarla a los dispositivos finales. Supongamos que no tenemos el control total de los dispositivos finales, por lo que estos pueden tener distintas versiones de ciertos componentes software esenciales para nuestra aplicación (dependencias o librerías). Cuando se procede al despliegue de la nueva actualización, se puede encontrar que en algunos de los dispositivos se produce un comportamiento inesperado o directamente la aplicación no funciona.

Otro problema al que enfrentarse es que los errores que se producen en una aplicación puedan afectar a otras, o incluso afecten a componentes principales del dispositivo, en cuyo caso habría que acudir al lugar en que se encuentre el dispositivo físicamente para sustituirlo o repararlo y conectarse a él directamente. Esto se conoce como problemas de aislamiento. Un ejemplo de esto sería que una aplicación haga una mala gestión de la memoria. Por ejemplo si la aplicación tiene un error y modifica parte de la memoria que está siendo utilizada por otra aplicación o por algún componente del kernel. En este caso dicha aplicación o componente realizaría operaciones inesperadas o directamente fallaría. Si se diese el caso de que la porción de memoria modificada se almacenase para utilizarse en futuras ejecuciones (por ejemplo, tras otro despertar del dispositivo), el problema se agravaría puesto que se producirá un comportamiento inesperado en el futuro.

Por todo esto, se ha hecho uso de contenedores en los dispositivos IoT. Al hacer uso de contenedores, las aplicaciones se empaquetan en una imagen junto con todas sus dependencias, por lo que se puede asegurar que la aplicación funciona de forma determinista en los dispositivos. Esto proporciona una solución al primer problema planteado. Los contenedores son una tecnología que aísla totalmente las aplicaciones unas de otras, ya que a cada imagen le corresponde un contenedor diferente, estando restringidos los recursos del sistema utilizados por cada contenedor, como por ejemplo la memoria y el espacio en disco. Esto implica que una aplicación no puede utilizar recursos que no pertenezcan al contenedor donde está siendo ejecutada. Quedando así resuelto el segundo problema.

Como último punto, pero no menos importante, existe una gran variedad de tipos de dispositivos IoT, cada uno con hardware y arquitectura de procesador diferente. Esto hace que crear una aplicación que sea portable entre distintos dispositivos sea todo un reto. Haciendo uso de contenedores, el programador, salvo ciertas excepciones, puede olvidarse de programar para un dispositivo concreto. Es el motor de contenedores (componente encargado de gestionar los contenedores en el dispostivo) el que se encarga de evadir estas diferencias, por lo que actúa como middleware, y por encima de él se ejecutan las aplicaciones. Se ofrece así una solución factible a este problema.

Además de todo esto, se proporciona una solución para gestionar el software que se ejecuta en los contenedores, se hace que la instalación y actualización de imágenes se realice de forma automática sobre todos los dispositivos de la red de una sola vez. Para realizar esta distribución se hace uso de una tecnología muy popular en la computación en la nube para la orquestación de contenedores llamada *Kubernetes*. Se ha creado un clúster de Kubernetes al que se han añadido los dispositivos y estos al unirse reciben órdenes de ejecución de aplicaciones. Estas órdenes a su vez pueden ser nuevas órdenes posteriores a la unión al clúster u órdenes previas a la unión del dispositivo si aún siguen vigentes. Kubernetes proporciona también una solución al problema de la escalabilidad ya que se pueden unir al clúster tantos dispositivos como se desee.

Para la creación de este clúster se han usado dispositivos Raspberry Pi como esclavos mientras que el maestro ha sido un ordenador portátil.

Durante la lectura de este documento puede evidenciarse que hemos tratado de adaptar herramientas muy potentes para la gestión de software utilizadas en la computación en la nube al ámbito del IoT.

En lo que sigue, tras presentar la situación actual y los objetivos del proyecto, se comentan las tecnologías empladas justificando su utilización. Se explican una serie de conceptos sobre contenedores y sobre Kubernetes. Se describe el diseño del sistema que se ha creado, así como las herramientas utilizadas y por útimo se describen las pruebas realizadas y se evaluan los resultados obtenidos.

1.1 Situación actual

Kubernetes y los contenedores se utilizan actualmente para desplegar servidores que se encarguen de procesar la gran cantidad de datos generados por los dispositivos, pero no se utilizan para gestionar las aplicaciones que ejecutan los propios dispositivos.

Un ejemplo en el que esta forma de gestión sería especialmente útil es en el desarrollo de ciudades inteligentes. Se podría pensar en los típicos carteles luminosos que se ven a diario en las paradas de los autobuses. En ellos aparece el tiempo que queda para la llegada del próximo autobús, este tiempo se podría consultar de cualquier servicio que posea los datos de por dónde cicua cada autobús. ¿Qué ocurriría si de repente se quisiera actualizar el programa que se ejecuta en ellas para, por ejemplo, mostrar también la previsión meteorológica? Habría que modificar el software que ejecuta para conectarlo a algún servicio de previsión meteorológica y tras ello actualizar todas las paradas. Con este sistema la actualización se limtaría a ejecutar un comando sobre el clúster.

Podría suponerse ahora que queremos añadir una nueva parada. Para ello sería necesario precargar el código que debe ejecutar antes de instalarla en la calle. Con este sistema tan solo habría que instalarla en la calle y automáticamente ejecutaría el mismo software que el resto de paradas.

Existe una gran diferencia entre dispositivos IoT y sistemas empotrados que debe quedar clara. Un dispositivo IoT, como su nombre indica, es un dispositivo que se conecta a internet para realizar alguna operación, ya sea almacenar datos, consultar un servicio, etc. Un sistema empotrado es un dispositivo que tiene un código internamente pero que no se conecta a internet ni necesita, por lo general, ser actualizado.

Se puede tomar como ejemplo una lavadora. Una lavadora es un sistema empotrado. Tan solo se selecciona el programa de lavado con la ruleta y se pone en marcha. Siempre hace lo mismo, ni se conecta a internet ni se actualiza.

Se podría tomar ahora el ejemplo de una lavadora moderna, que puede conectarse a un móvil e incluso actualizarse con nuevos programas de lavado. Este tipo de lavadoras si son un dispositivo IoT. Y es imaginable que la empresa que fabrica las lavadoras las actualice remotamente. Con este sistema las lavadoras podrían formar parte de un clúster y desde la empresa se podría actualizar el software de todas ellas con un único comando de forma remota y totalmente transparente al usuario real de la lavadora.

En cuanto a plataformas IoT para la gestión de los dispositivos, actualmente existen muchas, algunas de estas son IoTsens, AWS IoT, etc. Si bien la mayoría permiten la gestión del software de los dispositivos, no aportan los beneficios que nos ofrecen los contenedores en cuanto a aislamiento entre aplicaciones, seguridad, y determinismo en la ejecución en cualquier dispositivo. Otro problema que presentan las plataformas actuales es la escalabilidad. En estas plataformas se debe preparar el dispositivo y una vez preparado se añade manualmente a la misma para poder gestionarlo. Con el sistema que se ha desarrollado sigue siendo necesario preparar el dispositivo, pero una vez preparado se autoconfigura, se añade al clúster de forma automática y comienza a obedecer órdenes. Además, si una aplicación falla automáticamente se reinicia hasta que se ejecuta con éxito y podemos, en caso de configurarlo, obtener notificaciones cuando se produce un error utilizando por ejemplo Kubewatch.

1.2 Objetivos

El principal objetivo de este proyecto es crear un sistema de gestión de software para una red de dispositivos IoT que solucione los problemas de escalabilidad de la red y aislamiento, gestión y despligue de aplicaciones en los dispositivos utilizando contenedores que ejecutan las aplicaciones y Kubernetes para gestionar los contenedores que se ejecutan en los dispositivos. El uso de estas tecnologías permite actualizar de forma automática las aplicaciones de una sola vez en todos los dispositivos además de aportar una serie de ventajas como el aislamiento entre aplicaciones, seguridad, ejecución determinista en distintos entornos y abstracción de la heterogeneidad de dispositivos de una red.

Se enumeran a continuación el resto de objetivos:

- Optimizar el uso de recursos por parte de los contenedores utilizando tecnologías como balena-engine, debido a que los dispositivos IoT por norma general no poseen muchos recursos ni son demasiado potentes.
- Determinar qué tipo de dispositivos son los adecuados para utilizar esta tecnología y los recursos mínimos que necesitan. Este análisis se lleva a cabo durante toda la memoria según se comente acerca de elementos que sean críticos y jueguen un papel decisivo. Por lo general estas restricciones son mayoritariamente restricciones de capacidad, ya sea de memoria, de procesamiento, etc. Aunque también es fundamental el kernel del dispositivo.
- Examinar y comprender las particularidades que este sistema presenta respecto al uso tradicional de estas tecnologías en la computación en la nube aportando una adaptación. Es el caso del uso de sensores desde los contenedores, el uso de motores de contenedores alternativos, etc. Las herramientas que se utilizan ofrecen una gran cantidad de ventajas, pero no están pensadas para las restricciones en las que se incurre cuando se utilizan dispositivos IoT. Se emplean herramientas alternativas a las tradicionales, por ejemplo, se explica el uso de balena-engine en lugar de Docker como motor de contenedores para los dispositivos, ya que consume menos recursos. Esta adaptación al uso de herramientas alternativas es una de las principales fuentes de problemas en la creación del sistema.
- Demostrar de forma práctica que el sistema funciona y que resuelve los problemas planteados, creando un clúster de pruebas y dos ejemplos que se ejecutan en este clúster, mostrando el funcionamieinto de los ejemplos y el comportamiento del sistema completo, detallando paso a paso la creación del sistema, los scripts de automatización que se han escrito para crearlo y el despliegue de aplicaciones en él. En este clúster los esclavos son dispositivos RaspberryPi y el maestro una máquina virtual ejecutándose en un ordenador portátil. El esquema de alto nivel del clúster de pruebas es el siguiente:

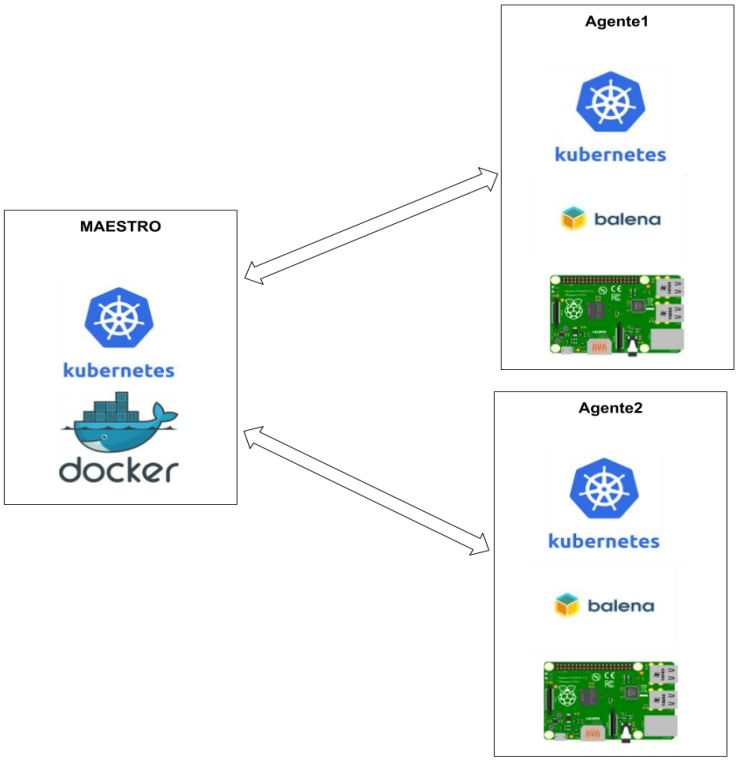

Figura 1-1: Esquema de alto nivel del clúster.

Además de los objetivos, comentar por último que se hace uso de herramientas de software libre. El software libre está ganando cada vez más fuerza y los proyectos de código abierto son cada vez más populares y reciben más apoyo por la comunidad de desarrolladores. De esta forma los proyectos se mantienen vivos gracias a los usuarios que contribuyen a ellos y no pertenecen a ninguna empresa en particular. Todo el mundo es libre de hacer uso de ellos, aprender de ellos y contribuir.

2 TECNOLOGÍAS

En este capítulo se presentan las tecnologías en las que se centra el diseño del sistema. Como se puede comprobar, algunas de estas tecnologías son actualmente muy utilizadas en la computación en la nube. Esto se debe a que son herramientas muy potentes que permiten una gestión de alto nivel realizando pocas operaciones. Todas las herramientas que se presentan están en auge y en continuo desarrollo, quedándoles aún un gran camino por recorrer antes de volverse obsoletas. A continuación, se comentan conceptos básicos sobre ellas y se les adapta a nuestro campo de aplicación, el IoT. Estas herramientas se combinan y configuran para que trabajen en conjunto con un único objetivo: gestionar el software de los dispositivos IoT con el mínimo esfuerzo posible.

Las tecnologías principales que se utilizan son Docker, balena-engine y Kubernetes. Tanto Docker como balena-engine son motores de contenedores, la diferencia es que balena-engine consume menos recursos por lo que la utilizamos en los dispositivos IoT, y utilizamos Docker en el maestro. Por otro lado, tenemos un clúster de kubernetes que es la base del sistema. Este clúster se encarga de conectar los dispositivos y darles órdenes.

Existen muchas tecnologías de comunicación para IoT. En este caso se utiliza Wifi, pero sería interesante indagar acerca del uso de LoRa o ZegBee.

En los siguientes apartados se presenta en el 2.1 conceptos sobre contenedores y lo relativo a estos y en el apartado 2.2 se comentan conceptos relacionados con Kubernetes.

2.1 Conceptos básicos sobre contenedores

Un contenedor es una unidad de software que empaqueta el código, todas sus dependencias, a veces un sistema operativo mínimo conocido como imagen base.

La ejecución de contenedores es posible gracias a ciertas características del kernel de Linux, como *cgroups.* Cgroups es una característica que limita y aisla el uso de recursos de una colección de procesos. Es decir, toma un conjunto de procesos, los aisla del resto de procesos del sistema y limita los recursos disponibles para estos.

A continuación, vamos a presentar una serie de conceptos generales y básicos sobre contenedores.

2.1.1.1 Imagen

Una imagen es un conjunto de archivos que constituye todo lo que será ejecutado dentro de un contenedor, se pueden interpretar como el conjunto de ficheros existentes dentro del contenedor, incluyendo un sistema operativo si se desea.

Las imágenes están organizadas en capas. Cada capa representa la diferencia entre lo que contiene y la capa anterior, siendo modificable tan solo la última capa. Esto es posible gracias a *OverlayFs,* que es un sistema de unión virtual de ficheros. Es decir, tenemos una serie de ficheros y directorios separados y OverlayFs los convierte en un único archivo, manteniendo el contenido real de estos separado. OverlayFs es utilizado por muchos motores de contenedores, pero concretamente tanto Docker como balena-engine lo utilizan.

Debemos tener en cuenta que todas las imágenes no están disponibles para todas las arquitecturas, y en nuestro caso necesitamos imágenes para ARM.

2.1.1.2 Registro de imágenes

El registro de imágenes no es más que un repositorio donde las imágenes se almacenan y cualquiera puede usarlas. Cuando ejecutamos un contenedor se busca la imagen primero en el registro local y, en caso de no encontrarla, se busca en el registro remoto que se tenga configurado. El más común, y el usado por defecto si no se modifica es DockerHub.

2.1.1.3 Redes de contenedores

Los contenedores utilizan una red virtual para conectarse entre ellos, y dentro de estas redes virtuales cada contenedor posee una dirección IP efímera, es decir, susceptible de ser cambiada y reutilizada por otro contenedor en cualquier momento. Podemos crear tantas redes de contenedores como queramos, pero si queremos desplegar un servidor en un contenedor necesitamos hacer un mapeo de un puerto del host al puerto/s del contenedor donde escuche el servicio, puesto que las redes virtuales no son accesibles desde el exterior de ningún otro modo. Para realizar el mapeo de puertos es necesario hacerlo en el momento de la creación del contenedor ya que no es posible realizar este mapeo una vez el contenedor se está ejecutando.

2.1.1.4 Volúmenes

Cuando un contenedor termina su ejecución se destruye, incluyendo la destrucción de todos los ficheros que se encontraban en su interior. Esto presenta un problema para las aplicaciones o servicios con estado. Para solucionar esto se utilizan los volúmenes.

Un volumen es un elemento especial que permite almacenar datos de forma persistente, así, cuando un contenedor se destruye, sus datos siguen disponibles para otro contenedor.

Los volúmenes los provee de forma automática el motor de contenedores cuando le son solicitados y además pueden ser mapeados a un directorio o fichero dentro del contenedor, lo que permite cargar ficheros y directorios del host dentro de un contenedor.

2.1.1.5 Dockerfile

Un fichero Dockerfile no es más que un fichero de texto que especifica las capas de las que estará compuesta una imagen. Generalmente cada línea corresponde a una instrucción y esa instrucción al ser ejecutada crea una nueva capa sobre la capa anterior. Cuando una imagen es reconstruida va leyendo capa por capa el Dockerfile y reconstruye tan solo a partir de la primera capa modificada. Tan solo puede modificarse la ultima capa de la imagen.

A continuación, se muestra un ejemplo de Dockerfile real:

```
FROM bitnami/node:10 AS builder
COPY package.json server.js /app
RUN npm install --prefix /app
FROM bitnami/node:10-prod
COPY --from=builder /app/package.json /app/server.js /app
COPY --from=builder /app/node_modules /app/node_modules
EXPOSE 8080
VOLUME /settings
RUN useradd -r -u 1001 -g root
nonroot
RUN chmod -R g+rwX /var/log
USER nonroot
WORKDIR /app
ENTRYPOINT ["node"]
CMD ["server.js"]
```

Cuadro de texto 2-1: Ejemplo de Dockerfile

En el Dockerfile anterior cada capa correspondería a cada una de las directivas que se escriben en mayúsculas. Se pueden apreciar directivas para copiar ficheros, cambiar de usuario, marcar un punto de montaje para un volumen, cambiar de directorio de trabajo, exponer un puerto, indicar el comando que se ejecutará al inciar el contenedor, etc. En cuanto al comando que se ejecuta al iniciar el contenedor, es la suma de lo escrito en el ENTRYPOINT y en la directiva CMD. La diferencia es que el ENTRYPOINT forma siempre parte del comando y se puede modificar por línea de comandos la parte correspondiente a CMD. En el ejemplo anterior se pueden modificar los parámetros de "node" pero siempre se ejecutará el comando "node".

2.1.2 Ejecución de contenedores: balena-engine y Docker

El uso de contenedores en cualquier dispositivo se basa en dos componentes fundamentales: "container-runtime" y "container-engine".

Existen muchas dudas acerca del significado de estos dos términos. Dependiendo del autor pueden variar levemente en cuanto a qué engloba cada uno. Se puede considerar que el "container-runtime" es el entorno de ejecución del contenedor (componentes y librerías software necesarias), mientras que el "container-engine" es el encargado de dar las órdenes de ejecutar contenedores y asignarles recursos del dispositivo. Normalmente cuando se instala un software de ejecución de contenedores se instalan los dos componentes y no es necesario instalarlos por separado.

Para ejecutar contenedores en los dispositivos IoT se usará de *balena-engine*, y como se verá más adelante en el maestro del clúster se usará *Docker.* Ambas soluciones son Open-source, lo que permite un mantenimiento y evolución rápida gracias a la comunidad de usuarios, y ambas están programadas en el lenguaje *Go*.

Docker es un motor de contenedores ("container-engine") de código abierto orientado a entornos cloud. Actualmente es el más utilizado y de hecho se está convirtiendo en un estándar en la computación en la nube. Por ello, balena-engine parte del código de Docker para adaptarlo a entornos IoT. Esto nos ha permitido que en el sistema convivan ambos sin grandes problemas. Balena-engine trata de optimizar los recursos utilizados por Docker para ser usado en dispositivos IoT, que suelen tener recursos bastante limitados. Esta adaptación ha sido posible gracias al proyecto Moby, que es un proyecto de código abierto, impulsado por Docker, para ensamblar sistemas de contenedores especializados, en este caso especializado en IoT.

Debido a que Docker estaba pensado para la nube, balena-engine elimina funcionalidades que si bien en entornos cloud erán indispensables, no tienen mucho sentido para IoT. Se enumeran a continuación las características de Docker que balena-engine elimina (para más información sobre cada característica consultar [9]):

- Docker Swarm (competencia de Kubernetes que acabó quedando de lado)
- Controladores de registro en la nube
- Soporte de Plugins
- Controladores de red superpuestos
- Almacenamiento no respaldado por boltdb

Las características principales de balena-engine que la hacen interesante para este proyecto son:

- Soporta seis arquitecturas de procesador diferentes (aarch64, armv5, armv6, armv7, i386 y x86_64), lo que hace que pueda utilizarse con muchos modelos de dispositivos, recordar que en este proyecto usaremos la placa Raspberry Pi cuya arquitectura es armv7.
- Ofrece la posibilidad de actualizar las imágenes de los contenedores mediante deltas, es decir, actualizar tan solo las capas de la imagen que han sido modificadas, lo que aporta un menor consumo de ancho de banda y por ello ahorro de batería en el dispositivo.

Tanto balena-engine como Docker serán los "container-engine" o motores de contendedores, y ambos se basan en *containerd*, un "container-runtime" estándar muy utilizado actualmente y que pertenece a la *Cloud Native Computing Foundation (CNCF)*. Containerd se instala en el dispositivo como un servicio más y podemos comunicarnos con él mediante una API y los sockets UNIX que utiliza, balena-engine y Docker hacen uso de esta API de forma transparente al usuario. Containerd se encarga de controlar el ciclo de vida del

contenedor.

Containerd a su vez utiliza *runc*. Runc es utilizado por containerd para lanzar y ejecutar los contenedores. Algunas empresas como Docker, Google y CoreOS crearon la *Open Container Initiative (OCI)* y liberaron su código de ejecución de contenedores, dando lugar a runc como implementación de referencia de la *OCI runtime specification* (una especificación estándar que determina la configuración, entorno de ejecución y ciclo de vida de un contenedor).

Cabe destacar que el uso de containerd restringe el ámbito de aplicación a dispositivos que soporten un kernel Linux. Esto se debe a que utiliza características propias de este para ciertas acciones, como el uso de espacios de nombres y asignación de permisos. En realidad, containerd está disponible también para Windows, pero nos centraremos en Linux ya que es mucho más utilizado en IoT y además es también de código abierto, lo que permite la creación de distribuiciones específicas para un dispositivo, usando por ejemplo, el proyecto Yocto.

Un dato interesante es que mientras se realizaba este proyecto, concretamente el 28 de Febrero de 2019, containerd se graduó de la CNCF, lo que significa que ya cuenta con suficientes usuarios como para ser mantenido por la comunidad.

Si se dividiese balena-engine en componentes se obtendría la siguiente estructura a groso modo:

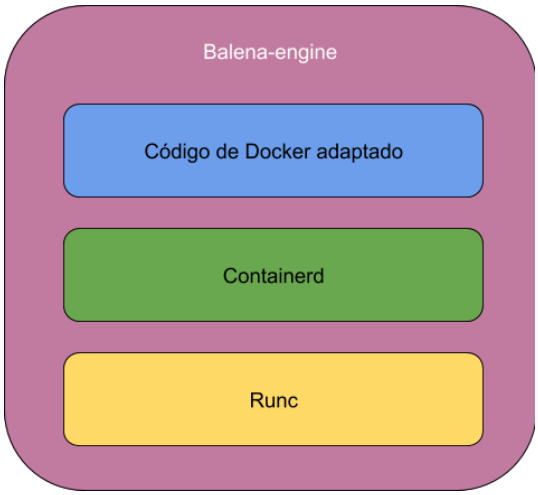

Figura 2-1: Estructura de balena-engine.

2.2 Automatización del despliegue en los dispositivos: Kubernetes

Hasta el momento se ha comentado lo que es un contenedor y las tecnologías que se utilizarán para ejecutar estos. El objetivo del trabajo consiste en gestionar el software y automatizar el despliegue de éste sobre los dispositivos. El simple hecho de utilizar contenedores no soluciona este problema, para resolverlo es necesario hacer uso de algún tipo de herramienta que se encargue de automatizar este proceso, y que mediante una única orden realice cambios en todos los dispositivos deseados. Esta herramienta será *Kubernetes*.

Kubernetes es un sistema de orquestación de contenedores de código abierto. Esto significa que es un sistema que permite automatizar el despliegue, escalar y gestionar aplicaciones que se ejecuten sobre contenedores. El problema que resolver aquí es que es un sistema pensado para nube y por tanto se hace necesario adaptarlo al IoT.

Gracias a kubernetes se podrá desplegar de una sola vez una aplicación sobre todos los dispositivos en uso (o una serie de ellos) o actualizar una existente. También permitirá reiniciar una aplicación en ejecución en cualquier dispositivo. Esto evitará tener que ir dispositivo por dispositivo instalándola o actualizándola.

Recordando, de los conceptos sobre contenedores, que estos se ejecutan en una red virtual, gracias a kubernetes se podrán conectar todos los contenedores, incluso los que están en nodos diferentes, sin necesidad

de mapear a puertos del host, a través de estas redes virtuales en caso de que la aplicación lo necesite. Esto aporta una gran seguridad adicional ya que estas redes virtuales no son accesibles desde el exterior a menos que se mapee algún puerto del host.

Kubernetes ofrece características muy interesantes como por ejemplo un mecanismo llamado Rolling-update. Este mecanismo consiste en la actualización ordenada de los contenedores que se están ejecutando en los nodos, de manera que, se van actualizando en tandas de n contenedores prefijadas. Cuando los primeros n contenedores están listos para ser usados y sin errores se actualizan otros n, y así hasta actualizarlos todos.

Para hacer uso de kubernetes simplemente hay que utilizar un cliente que haga uso de su API y esté conectado a la API que expone el clúster. Absolutamente todo lo que permite kubernetes se realiza a través de su API.

Kubernetes suele utilizarse sobre clusters en la nube que proporcionan las empresas que ofrecen servicios de computación en la nube. En el caso que nos ocupa, se creará un clúster propio haciendo que los dispositivos IoT sean miembros de este.

La estructura típica del clúster es maestro esclavo, normalmente varios maestros que reparten la carga computacional entre los esclavos en función de ciertos parámetros. En este caso, como se verá más adelante, se tendrá un único maestro y tantos esclavos como dispositivos IoT.

El principal rival de kubernetes ha sido tradicionalmente Docker Swarm. Docker Swarm es la plataforma de orquestación de contenedores propia de Docker. Pese a ser también propiedad de Docker, Kubernetes ha ganado la batalla de la orquestación y es sin duda el más utilizado. Ambos presentan ciertas ventajas e incovenientes, pero la principal razón de la imposición de kubernetes en el mercado se debe al apoyo de la comunidad. El proyeceto de kubernetes es mucho más apoyado y mantenido por lo que evoluciona más rápido y se ha convertido en el estandar. Debido a ello, utilizaremos kubernetes y no Docker Swarm. Aclarar que Docker y Docker Swarm no tienen nada que ver. Docker es el motor de contenedores mientras que Docker Swarm es tan solo una plataforma de orquestación que pertenece a la misma empresa.

Por último, comentar que Kubernetes no es una tecnología nueva como se suele pensar, lleva en uso más de 15 años, aunque era un sistema interno de Google hasta que se hizo público hace 5 años.

2.2.1 Conceptos básicos sobre kubernetes

A continuación, se presentarán una serie de conceptos básicos sobre kubernetes y posteriormente se discutirá la necesidad de adaptarlos a nuestro ámbito de aplicación.

A partir de este momento nos referiremos a estos conceptos por sus nombres en inglés, para evitar confusiones.

Cada concepto de los explicados a continuación se corresponde con un objeto de kubernetes. Existen más tipos además de los presentados en este apartado, estos son los más básicos y entendiendo estos entenderemos el resto fácilmente. Cuando necesitemos otros objetos serán comentados sobre la marcha. El objetivo de este apartado es cimentar los conceptos básicos.

2.2.1.1 Pods

Los Pods son la unidad mínima de computación en kubernetes. Un pod representa un conjunto de uno o más contenedores que comparten almacenamiento, una única dirección IP y espacio de puertos.

Debemos destacar que los Pods son efímeros, es decir, pueden crearse y eliminarse en cualquier momento, por lo que una aplicación que se ejecute sobre kubernetes debe estar diseñada para soportar esto. Básicamente debe estar preparada para separar el estado de la ejecución.

Dentro de un Pod también Puede haber volúmenes, que los veremos a continuación.

2.2.1.2 Volúmenes

Para separar la ejecución del estado, el estado debe almacenarse de forma separada. Para ello existen los volúmenes. El concepto de volumen es el mismo que se explicó en los conceptos sobre contenedores. En este

caso, un mismo volúmen se puede montar dentro de varios Pods.

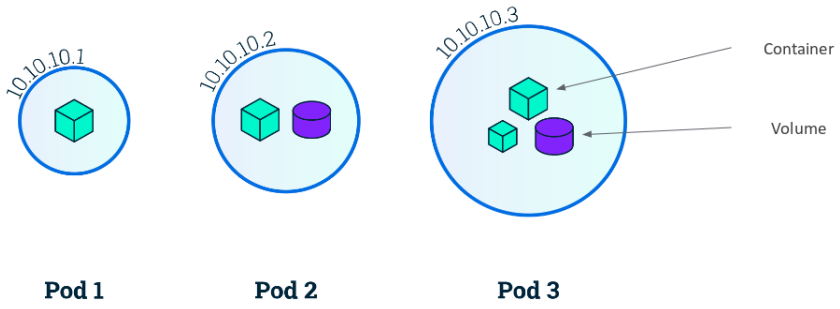

Figura 2-2: Pod

2.2.1.3 ReplicaSet

El ReplicaSet es un objeto que se encarga de asegurar que cierto número de Pods están en ejecución en un momento dado, esto permite tener un tiempo nulo de caída de los servicios. Además, aporta mecanismos automáticos para crear nuevos Pods cuando el número existente es menor que el establecido, y nos aporta también la posibilidad de escalar el número de Pods dinámicamente.

Para seleccionar los Pods a los que afecta un ReplicaSet se utilizan etiquetas. Simplemente en el fichero YAML de definición del ReplicaSet se da una etiqueta al Pod y en la sección "matchlabels" del ReplicaSet se establece la etiqueta que se le dio al Pod. La etiqueta no es más que un par clave-valor, que puede ser lo que nosotros queramos y la usamos para identificar recursos o conjuntos de recursos.

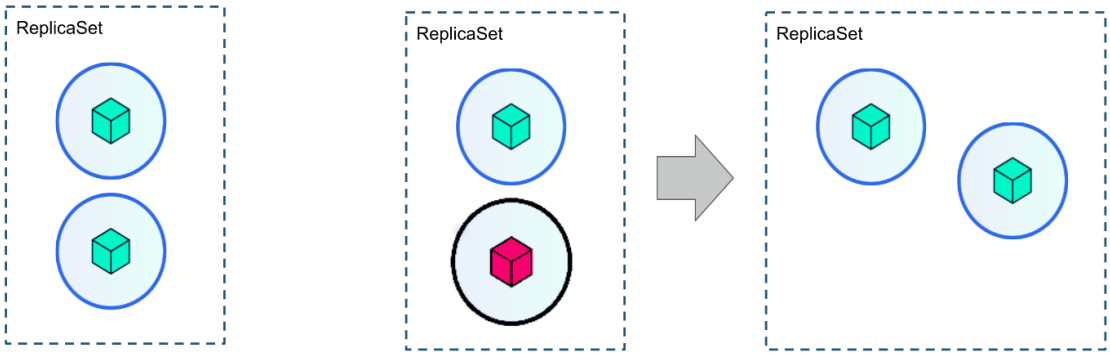

Figura 2-3: Replicaset

2.2.1.4 Deployments

Un Deployment es un concepto de más alto nivel que un ReplicaSet. El Deployment permite gestionar un conjunto de ReplicaSets, y es el encargado de gestionar el mecanismo de Rolling-Update que se comentó. También permite realizar Rollbacks, es decir, devolver todo el Deployment a un estado anterior en caso de errores al realizar una actualización.

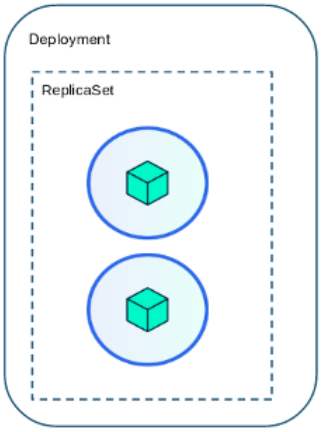

Figura 2-4: Deployment

2.2.1.5 Services

Un service, no es más que una abstracción que define un conjunto de Pods que proveen de un micro-servicio. Como se ha comentado los Pods son efímeros, por lo que no tendría sentido conectarnos directamente a un pod, puesto que en caso de que el pod se destruya perderíamos la conexión. Esto se soluciona con los services, que son un punto permanente al que nos podemos conectar y este, automáticamente, redirecciona a un pod que esté en ejecución. Además, en caso de que el pod al que ha redireccionado se destruya, nos conectará de forma totalmente transparente a otro pod. Aquí es donde entra en juego la importancia de separar el estado de la ejecución, ya que al ser redireccionado a otro pod debemos conservar el estado aunque se haya terminado la ejecución del pod al que estábamos conectados.

Por otra parte, los services nos permiten exponer al exterior los servicios que se están ejecutando en el clúster, de forma que las conexiones entrarán a través de un service, e irán a los Pods. Así basta con saber la IP del service. Aclarar que los services tienen direcciones IP virtuales del espacio de direcciones virtual de kubernetes, por lo que no es posible acceder a la IP de un service desde el exterior del clúster. Para ello, al igual que pasaba con Docker, es necesario mapear un puerto del host al puerto e IP del service.

Los services se implementan gracias a iptables, es *Kube-proxy,* un componente del plano de control de kubernetes el encargado de crear las reglas necesarias. Cuando se reciben conexiones, gracias a kube-proxy se redirigen al Pod correspondiente mediante las reglas de iptables.

Existen tres tipos de services fundamentales en Kubernetes:

- ClusterIP: este tipo sirve tan solo para ser utilizado por Pods para conectarse a otros Pods, es decir, es totalmente inaccesible desde fuera del clúster.

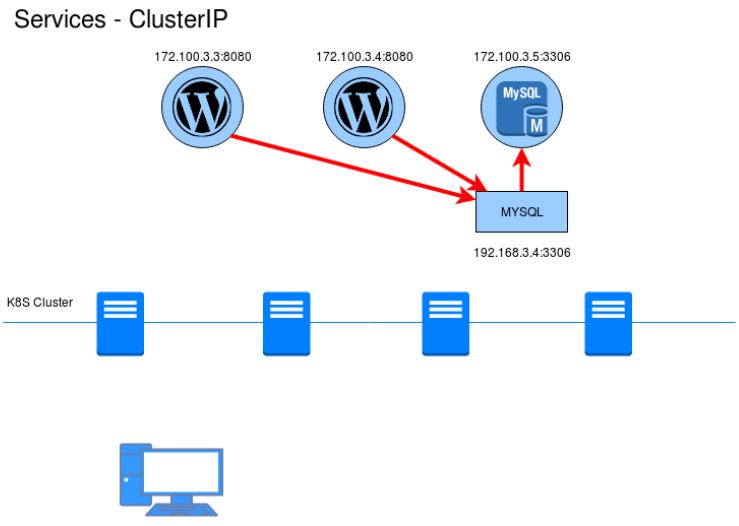

Figura 2-5: Service - ClusterIP

- NodePort: este servicio provee de un punto de entrada al clúster desde el exterior, mapeando automáticamente un puerto del host al puerto del servicio.

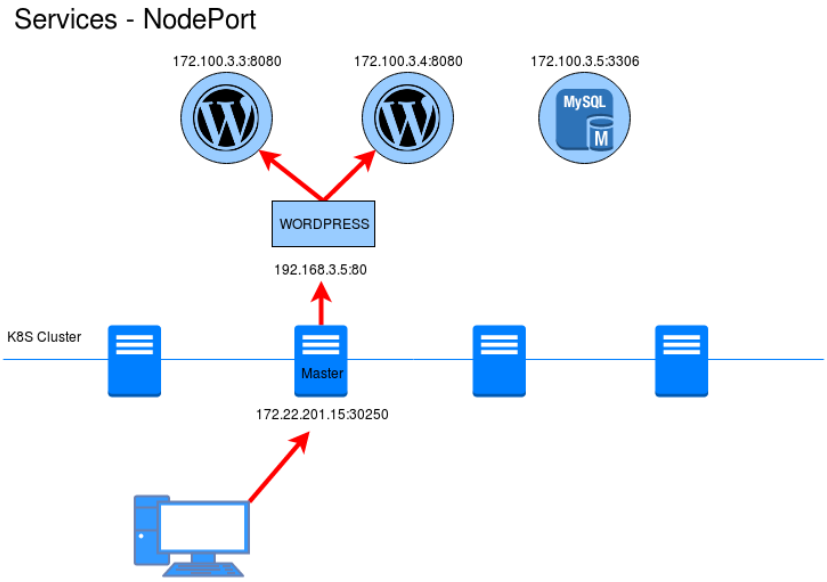

Figura 2-6: Service - NodePort

- LoadBalancer: como su nombre indica este tipo de servicio se encarga de hacer un balanceo de carga entre varios nodos del clúster cuando le llegan las peticiones externas.

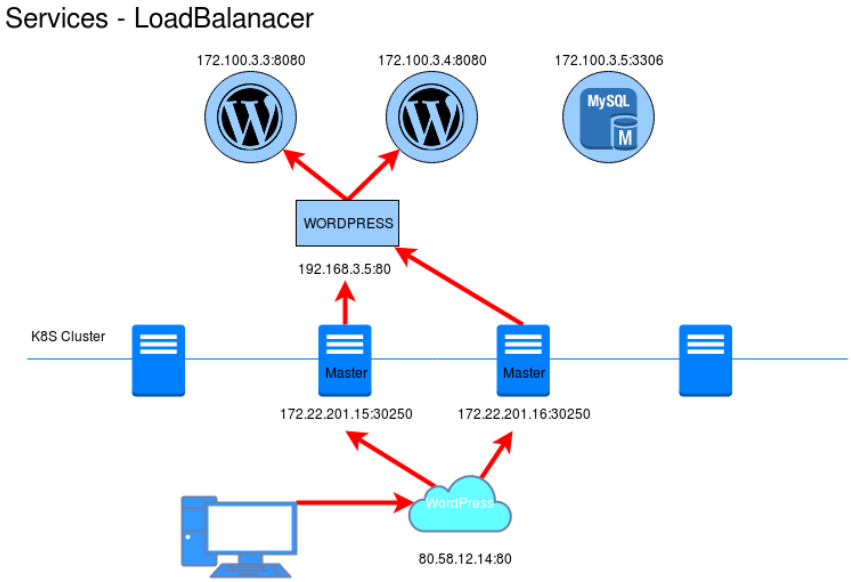

Figura 2-7: Service - LoadBalancer

Aunque hemos utilizado imágenes con los simbólos de Wordpress y MySQL no tenemos intención de desplegar nada por el estilo en el clúster ya que nos enfocamos a dispositivos IoT, hemos utilizado esas imágenes solo para explicar el concepto. Esos son el tipo de servivios que se despliegan en un entorno cloud.

A través de los Services las conexiones irán a un Pod u otro en función de las etiquetas que se pongan a los Pods.

En la siguiente ilustración se muestra la estructura del clúster. En ella, los usuarios representan personas qe

acceden a servicios y el Developer/Operator representa el gestor del clúster que realiza operaciones sobre él, en este caso nosotros. En nuestro ámbito puede que tengamos o no usuarios, dependiendo de si los dispositivos IoT exponen algún servicio. Por ejemplo, podríamos exponer un servicio de consulta de la temperatura de un termómetro para que usuarios externos la consulten, y mediante un control de acceso determinar en qué dispositivo se quiere consultar, por ejemplo, determinando el dispositivo mediante la URL en una petición GET. O podríamos no tener usuarios si simplemente lo utilizamos para gestionar software en la red de dispositivos.

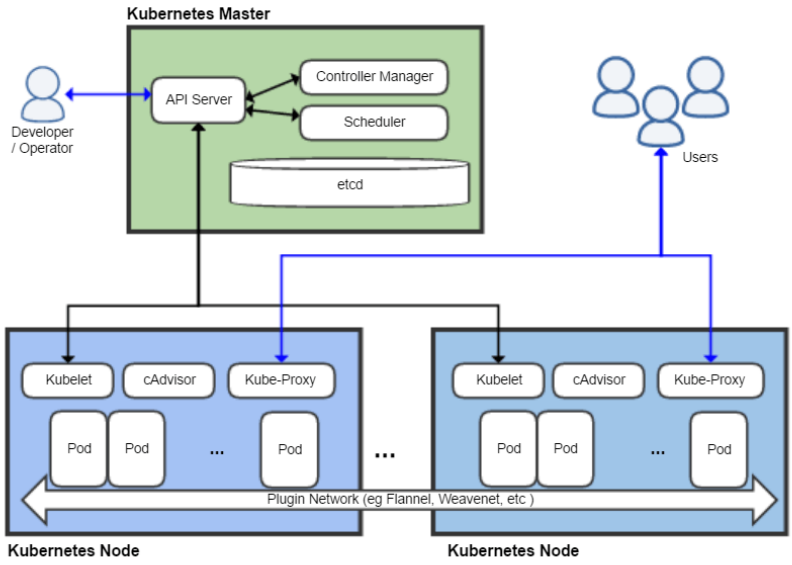

Figura 2-8: Acceso a servicios a través de kube-proxy

Los componentes del maestro (API server, Controller Manager, Scheduler, Etcd) serán comentados más adelante.

2.2.1.6 ConfigMap

Un ConfigMap, como su nombre indica, es un objeto que nos permite almacenar datos de configuración del clúster como por ejemplo ficheros de configuración, datos iniciales para cargar en una aplicación, etc.

2.2.1.7 Secret

Como su nombre indica, los Secrets son objetos para almacenar información confidencial en el clúster para ser usada por las aplicaciones, por ejemplo, las claves para conexión con una base de datos. Aunque cabe destacar que los datos de un Secret no están cifrados, tan solo están codificados en base64 por lo que su contenido puede decodificarse con un simple comando.

2.2.2 Uso de kubernetes

Por último, comentar que, kubernetes se puede utilizar principalmente de dos formas. La primera mediante una pequeña aplicación web que permite crear los objetos de forma intercativa, aunque ésta no es la manera habitual. La segunda, y la que realmente se suele utilizar, creando los ficheros YAML que definen cada objeto y posteriormente creando los objetos a través del cliente de línea de comandos llamado *Kubectl,* haciendo uso de los ficheros. Los objetos JSON se envían a la API REST que expone el maestro del clúster y a través de ella se realiza toda la gestión.

Un fichero YAML no es más que un envoltorio de un fichero JSON. En el momento de la creación de los objetos, Kubectl se encargará de transformar los ficheros YAML a objetos JSON.

A continuación, se presentan tres ejemplos de YAML, cada uno correspondiente a un objeto de Kubernetes.

Observar como cada objeto engloba al anterior.

```
apiVersion: v1
kind: pod
metadata:
    name: mongo
spec:
    containers:
    -   image: bitnami/mongodb
        name: mongo
```

Cuadro de texto 2-2: YAML de definición de un pod

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
    name: mongo
    namespace: default
spec:
    strategy:
        type: RollingUpdate
    replicas: 2
    selector:
        matchLabels:
             app: mongo
    template:
        metadata:
          name: mongo
    spec:
        containers:
        -  image: bitnami/mongodb
           name: mongo
```

Cuadro de texto 2-4: YAML de definición de un deployment

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
    name: mongo
    namespace: default
spec:
     replicas: 2
     selector:
         matchLabels:
              app: mongo
    template:
        metadata:
          name: mongo
    spec:
        containers:
        -  image: bitnami/mongodb
           name: mongo
```

Cuadro de texto 2-3: YAML de definición de un ReplicaSet

2.2.3 Tecnologías de creación de un clúster: Kubeadm, Kubelet y Kubectl

Normalmente cuando se desea hacer uso de un clúster de Kubernetes se subcontrata este servicio a proveedores como Google (GKE) o Amazon (AKS). Ese tipo de servicio es orientado a la computación en la nube para desplegar otros servicios. Por ello, vamos a crear nuestro propio clúster con las tecnologías descritas, tratando de adaptarlo todo lo posible a dispositivos IoT y en que cada nodo será un dipositivo.

Existen varias herramientas para crear un clúster de kubernetes, una de las que más se trabaja en desarrollar actualmente es *Kubeadm* y dada su perspectiva de futuro será la que utilizaremos. Kubeadm se encarga de crear en el maestro de iniciar el clúster y en los esclavos de conectarse a el.

Además de Kubeadm es necesario utilizar otras dos herramientas auxiliares, una encargada de ejecutar los Pods y contenedores llamada *Kubelet* y un cliente que haga uso de la API de kubernetes para comunicarnos con el clúster, llamado *Kubectl*.

Kubelet hace uso de dockershim como CRI (Container Runtime Interface) por defecto. Existen otras opciones de CRI, por ejemplo CRI-O, que es una implementación de CRI basado en la OCI (Open Container Initiative). Sin embargo, dockershim es el utilizado por Docker, y como se ha visto, balena-engine es una adaptación de Docker, por lo que ambos motores de contentenedores tendrán el mismo CRI, y, por lo tanto, se podrá hacer uso de kubelet sin modificar el CRI que utiliza, que es una tarea compleja.

Se continuará hablando de estos tres componentes durante el resto de esta memoria.

Las características recomendadas para que un dispositivo pueda ejecutar los contenedores y hacer uso de Kubernetes al mismo tiempo sin problemas son:

- 2GB de memoria RAM
- 2CPUs
- Conectividad de red
- Nombre único del host
- Desactivar el SWAP, el SWAP permite mover procesos de memoria RAM a disco y viceversa, por lo tanto, se pueden ejecutar procesos que en conjunto ocupan más memoria de la disponible. Kubeadem requiere que esta característica esté descativada.

Resulta por tanto que, de nuevo, se reduce el número de dispositivos capaces de formar parte de nuestro clúster puesto que muchos de los dispositivos IoT actuales no poseen estas características. Sin embargo, en nuestro caso se han realizado pruebas con una placa Raspberry PI que no cumple el requisito de memoria RAM (la RaspberryPi tiene solo 1GB) y el resultado ha sido favorable. Por lo que se concluye que estas características **no son obligatorias** y que un dispositivo ligeramente inferior puede soportarlo.

2.2.4 Kernel y Sistema Operativo

Actualmente existen varios proyectos destinados a la creación de kernels y sistemas operativos para IoT. Dos de los más relevantes son el proyecto Yocto y el proyecto Zephyr, ambos enfocados en la portabilidad entre distintas arquitecturas.

El proyecto *Zephyr* es un proyecto de la Fundación Linux (*The Linux Fundation*) en colaboración con otros organismos y empresas del sector como, por ejemplo, Intel. El proyecto Zephyr nos permite crear un sistema operativo de tiempo real (*RTOS*) para dispositivos embebidos e IoT. Se permite el escalado del SO añadiendo los módulos deseados antes de su construcción o eliminando módulos base que no se precisen. Este proyecto se centra en soportar múltiples arquitecturas hardware (soporta algo más de 150 placas diferentes) y está pensado para dispositivos con **recursos muy limitados**.

Por otro lado, el proyecto *Yocto* es un proyecto de código abierto, también perteneciente a la Fundación Linux en el que también participan grandes empresas tecnológicas como Facebook e Intel. Este proyecto está centrado en la creación de **distribuiciones Linux mínimas** para dispositivos embebidos e IoT. También soporta diversas arquitecturas harware (aunque menos que proyecto Zephyr) y permite a partir de un paquete base ir añadiendo módulos para construir una distribución Linux que se adapte a las necesidades de cada situación. A diferencia de Zephyr se parte de una base que no es modificable y solo se puede añadir modulos a ella. El paquete base lo elige el usuario y puede ir desde lo mínimo para que un dispositivo inicie (boot) hasta el kernel de Linux más avanzado.

En nuestro caso, como comentaremos más adelante, necesitamos hacer uso de un kernel Linux. Además, los dispositivos de recursos muy limitados a los que está orientado el proyecto Zephyr normalmente realizan una única función, por lo que no suelen necesitar gestionar varias aplicaciones. Por ello, se considera que el proyecto Yocto sería el apropiado para construir una distribución mínima específica para nuestro sistema.

Existen multitud de SO creados a partir del proyecto Yocto utilizados a dia de hoy en dispositivos en producción, que serían compatibles con nuestro sistema, por ejemplo, BalenaOS. Construir una distribución propia es útil si nuestros dispositivos tienen unos recursos muy limitados y necesitamos optimizar su uso al máximo, cosa que suele ocurrir en el IoT, y por ello hablaríamos de la creación de una distribución con tan solo los componentes esenciales para que el dispositivo fuese capaz de ejecutar contenedores. BalenaOS por ejemplo es una distribución mínima con este propósito. No se abordará la construcción del sistema operativo en este proyecto, simplemente se trata de dar una visión general sobre cómo avanzan las tecnológias en este ámbito y cómo Linux va ganando terreno en el IoT.

En nuestro caso, resulta irrelevante el sistema operativo que utilicen los dispositivos, incluso podrían coexistir dispositivos con diferentes sistemas operativos y diferentes arquitecturas de procesador (como ocurrirá en nuestro sistema entre el maestro y los esclavos) siempre que sean Linux. Ya que no se dispone de dispositivos IoT reales con las características mencionadas, se hace uso de la placa Raspberry pi, que utiliza Raspbian como sistema operativo, una distribución lijera de 1,3GB basada en Debian y que por tanto posee un kernel Linux. Somos conscientes de que para un dispositivo menos avanzado esto puede ser mucho espacio, en cualquier caso, no necesitamos la mayoría de funcionalidades que vienen con Raspbian por defecto, por lo que esta cifra se reduciría utilizando una distribución especializada.

Teniendo en cuenta esto, el presenta proyecto se centra en dispositivos con un cierto nivel de complejidad, esencialmente los que sean capaces de ejecutar un kernel de Linux mínimo. Excluyendo del mismo los dispositivos más básicos que no lo soporten. Esto no supone un gran problema ya que actualmente tres cuartas partes de los dispositivos IoT existentes se basan en un kernel Linux, pero hay que tener presentes que otra gran parte de dispositivos son demasiado simples para el uso de este sistema. Esta restricción viene impuesta por balena-engine y, concretamente, por containerd.

En resumen, nos centraremos en dispositivos IoT y, dentro del ámbito del IoT, en dispositivos que trabajan con un kernel Linux (el término kernel Linux engloba de forma convencional tanto al kernel como al sistema operativo).

2.3 Resumen del capítulo

En este capitulo se han introducido las tecnologías que se usarán durante el resto de capítulos. Las principales serán balena-engine, Docker y Kubernetes, y todo lo que estas engloban. Tanto balena-engine como Docker son motores de contenedores que sirven para ejecutar los contenedores en los dispositivos (balena-engine) y en el maestro del clúster (Docker). Por otra parte Kubernetes permite agrupar los contenedores y gestionar el despliegue completo, escalarlo, actualizarlo, etc.

Se han explicado resumidamente conceptos básicos sobre contenedores y Kubernetes, necesarios para comprender el resto del proyecto. Se han introducido también las herramientas que nos permiten crear un clúster que son Kubeadm, Kubectl y Kubelet. Por último, se han comentado dos proyectos de creación de kernels y sistemas operativos para dispositivos IoT, y se ha llegado a la conclusión de que en nuestro sistema solo podrán existir dispositivos con un kernel Linux.

En el siguiente capítulo se entrará en mayor profundidad en estos conceptos y se aplicarán al diseño del

sistema, para posteriormente pasar a la práctica, crear un clúster y desplegar aplicaciones de pruebas.

3 DISEÑO DEL SISTEMA

En este capítulo se va a presentar detalladamente el diseño del sistema. El sistema se basará en una arquitectura maestro-esclavo. Por lo que se presentarán el diseño del maestro y el diseño de los esclavos por separado. En el caso de los esclavos, al estar centrados en dispositivos IoT, se profundizará en el dispositivo, haciendo hincapié en la arquitectura hardware y en el sistema operativo, además de las capas superiores. Se hace referencia a los dispositivos esclavos como "**Agentes**". El maestro, dará ordenes a los agentes. En él, tendremos los componentes encargados de administrar el clúster. El maestro expondrá una API que podrá ser utilizada para dar órdenes al clúster por cualquier máquina externa. El esquema final del clúster que se ha creado para las pruebas es el siguiente:

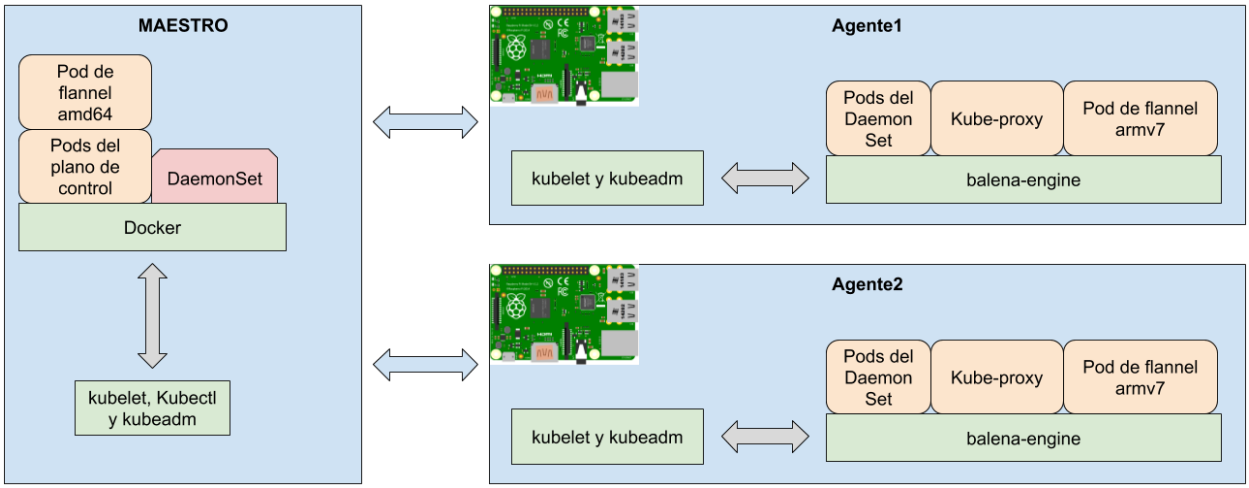

Figura 3-1: Esquema de componentes de los dispositivos en el clúster de pruebas

Todos lo elementos representados en su interior serán explicados en este capítulo. Concretamente, en el apartado 3.1 se presentará el diseño de los agentes y se comentará acerca de las aplicaciones que podrán ejecutar y del uso de sensores. En el 3.2 se explicará el diseño del maestro y en el 3.3 se explicará el proceso de creación del clúster y características de este como el direccionamiento interno, la auto-configuración, el despliegue de aplicaciones, la conexión con el clúster para darle órdenes, etc. En el apartado 3.4 se crearán dos imágenes para dos aplicaciones de prueba, siguiendo un proceso de optimización hasta lograr que sean lo más pequeñas posible.

3.1 Diseño de los agentes

La funcionalidad principal de los agentes será ejecutar aplicaciones. Estas aplicaciones deben instalarse y actualizarse de forma remota desde el maestro. A continuación, se muestra en relación a las tecnologías descritas, la estructura de un agente:

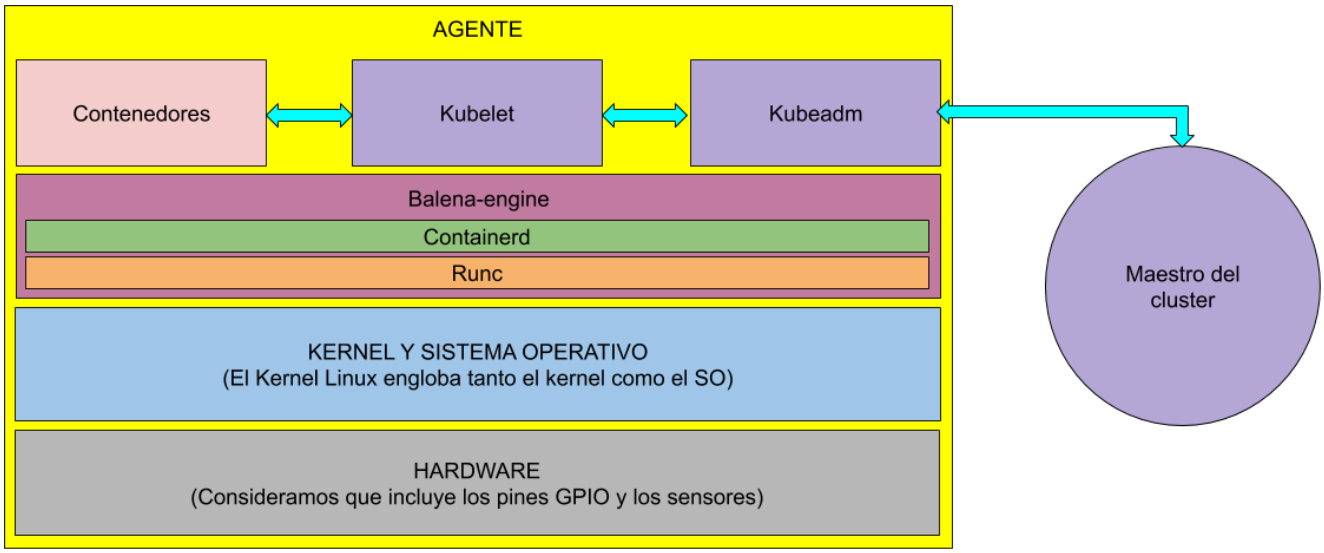

Figura 3-2: Estructura de un agente

Como ya se ha mencionado, los agentes utilizados son dispositivos Raspberry Pi, que ejecutan balena-engine como motor de contenedores y además ejecutan Kubeadm y Kubelet para conectarse al clúster, no siendo necesario en ellos kubectl puesto que no hacen peticiones al API-server.

3.1.1 Separación del estado y la ejecución de una aplicación

Las aplicaciones que se podrán ejecutar sobre los dispositivos serán variadas, pero siempre las serán ejecutadas dentro de un contenedor que utilizará balena-engine como motor de contenedores. Un aspecto a tener en cuenta es la limitación de memoria que suelen tener los dispositivos IoT. Para ello, al crear las imágenes que ejecutarán se debe tener en cuenta que puede no ser una buena práctica el usar un sistema operativo completo dentro del contenedor. Una opción sería utilizar software compilado, lo que nos permite que la imagen ocupe tan solo el tamaño del fichero binario ejecutable, creando las imágenes con la instrucción “FROM scratch”. Si en cualquier caso fuera preciso hacer uso de un software interpretado, por ejemplo una aplicación basada en node.js, sería necesario ejecutar dentro del contenedor un sistema operativo que ejecutase node.js. En estos casos se puede recurrir a imágenes mínimas y partir de ellas en el Dockerfile con la instrucción “FROM” para agregar lo que necesitemos, teniendo en cuenta que en este caso la imagen tendrá un tamaño mayor, o recurrir a un compilador para este código interpretado. Se profundizará acerca de este tema en el apartado 3.4 Creación de imágenes para los dispositivos.

Haciendo uso de Dockerfile se puede crear una imagen prácticamente de cualquier aplicación porque el sistema que se está diseñando soportaría muchas de las aplicaciones de IoT. La única adaptación que debería hacerse, en caso de que no haya sido diseñada así, es la ya comentada separación del estado y ejecución, para que fuese posible su despliegue en nuestro clúster.

Un ejemplo que ilustra esto es el siguiente. Suponiendo que tenemos una serie de dispositivos repartidos por una ciudad, que miden el nivel de CO2 periódicamente. La aplicación al ejecutarse no depende del nivel de CO2 que midió en la ejecución anterior por lo que no habría ningún problema en crear una imagen de esa aplicación y distribuirla con nuestro sistema a todos los dispositivos. La aplicación al realizar la medida guardaría en un volumen el nuevo nivel medido, para que al destruirse el contenedor tras la toma de la medida no se pierdan los datos. Se podría montar el volumen gracias a kubernetes como un SharedVolume, esto trae consigo otra ventaja y es que, al almacenar los datos en un volumen compartido del clúster, ese volumen puede ser utilizado por todos los Pods que se ejecutan en todos dispositivos lo que se traduce en que tenemos todos nuestros datos centralizados sin necesidad de realizar ninguna otra operación.

El ejemplo anterior es útil para ilustrar cómo se separa el estado de la aplicación, pero se debe tener en cuenta

que este ejemplo es válido si entre cada medida existe un lapso de tiempo del orden de horas, ya que sería necesario que el maestro estuviese dando periódicamente las órdenes a todos los dispositivos consumiendo bastante ancho de banda y por ello batería. Si por ejemplo se tomasen las medidas cada 30 segundos sería más eficiente que se distribuyese una sola vez la imagen y que esta se quedase en ejecución permanentemente realizando periódicamente la medida, evitando así la sobrecarga de instrucciones del maestro a los esclavos. En caso de que la aplicación anterior dependiese de datos de la ejecución anterior estos datos deberían cargarse desde un volumen, de forma que la aplicación se ejecuta siempre igual, pero dependiendo de los datos que toma del volumen, y así quedarían separados el estado de la ejecución.

En el ejemplo se ha supuesto que se miden los niveles de CO2, para ello es necesario disponer de un sensor. Se debe cuestionar ahora si es posible hacer uso de sensores del host desde un contenedor que se ejecuta de forma aislada del resto de procesos en el dispositivo. Este punto se analiza en el próximo apartado.

3.1.2 Uso de sensores o hardware desde un contenedor.

Para hacer uso de sensores, y en definitiva para controlar circuitos físicos es necesario tener acceso a los pines del dispositivo, en nuestro caso a los pines de la Raspberry Pi.

Para ello, se necesita hacer uso de alguna herramienta que se encargue de gestionar este acceso. Se usará *GPIO* (*General Purpose Input/Outpu*t), que como su nombre indica es un sistema de entrada/salida de propósito general y se encargará de gestionar los pines. Se utilizará también una librería llamada *WiringPi*, que permite acceder al sistema GPIO cómodamente desde varios lenguajes de programación (C, Python, Ruby, etc.)

Pero esto no es sufiente. Recordando, un contenedor se ejecuta aislado del resto de procesos del dispositivo. Si se crea un contenedor con un programa que haga uso de WiringPi y se ejecuta, veremos cómo realmente no tiene acceso a los pines de entrada/salida. Esto es debido a que un contenedor por defecto se ejecuta en modo no privilegiado y por tanto no tiene acceso a ningún periférico del dispositivo.

Si ejecutamos el contenedor con la opción `--privileged`, tendríamos acceso al fichero `/dev/mem` (que contiene un mapa de la memoria principal del dispositivo usado por los pines GPIO), entonces se podría acceder a todos los pines del dispositivo sin problema, pero también a otros ficheros que no son necesarios, con permisos de lectura y escritura. Esto presenta un serio problema de seguridad puesto que cualquier persona que consiga acceder al contenedor podrá tener acceso y modificar ficheros importantes para el funcionamiento del dispositivo.

Para solucionarlo, en el momento de la ejecución, en vez de hacerlo en modo privilegiado, se mapea el fichero `/dev/gpiomem` del dispositivo al dichero `/dev/gpiomem` del contenedor. Este fichero es un fichero especial con el único propósito de dar acceso a los pines GPIO sin necesidad de ser un usuario con privilegios y por tanto si alguna persona externa accede al contendor como mucho tendrá acceso a los pines GPIO. Esto ser hace utilizando el comando `balena-engine run --device /dev/gpiomem <imagen>`. Se hace uso de “device” ya que este fichero no es un fichero normal sino un fichero utilizado para acceder al hardware, y así la aplicación que se ejecuta en el contenedor podrá acceder al hardware sin ser modificada.

De esta manera el uso de sensores es aplicable cuando se ejecutan los contenedores directamente con el comando `balena-engine`. Pero en nuestro sistema final, los contenedores serán iniciados por Kubelet a través del clúster. El problema que surge ahora es que en Kubernetes no se puede modificar el comando que se ejecuta internamente para iniciar los contenedores. El maestro da las órdenes y el agente de Kubelet existente en cada dispositivo es el que ejecuta los contenedores. Es necesario por tanto otra forma de mapear los pines GPIO.

Se ha investigado bastante acerca del tema y hecho diferentes pruebas con distintos componentes de Kubernetes hasta encontrar la forma de conseguir esto. Para ello, se debe montar un fichero del nodo en el contenedor haciendo uso de la directiva *hostPath.* Esta directiva identifica un tipo de volumen especial en Kubernetes que permite montar una ruta (*path*) concreto del host que aloja el nodo en una ruta concreta del sistema de ficheros del contenedor.

Llegar a esta conclusión no fue simple ya que aquí se encontró una contradicción con la documentación oficial, y es que, en [30] se puede leer que no es posible hacer uso de esta directiva en clústers con más de un nodo. En nuestro caso, se ha hecho la prueba con nuestro clúster de un maestro y dos esclavos y se ha

comprobado que se puede y que, de hecho, funciona correctamente. Un ejemplo de cómo se realiza el montaje de estos volúmenes que referencian rutas concretas es el presentado a continuación, este fragmento de fichero es de un ejemplo creado para nuestros despliegues de pruebas y lo se verá más adelante:

```
- name: my-ds
   image: docker.io/miguelaeh/sensorluzcompilado
   tty: true
   securityContext:
     privileged: true
   volumeMounts:
     - mountPath: /dev/gpiomem
       name: vol
 volumes:
  - name: vol
    hostPath:
      path: /dev/gpiomem
      type: File
```

Cuadro de texto 3-1: Volumen para lectura de pines GPIO.

El fragmento anterior engloba la definición de un Pod y del volumen asociado que monta para acceder a los pines GPIO.

3.2 Diseño del maestro

La función principal del maestro será dar órdenes como maestro del cluster de kubernetes al resto de dispositivos, realizando la gestión de los contenedores a través de kubelet que se ejecuta en cada uno de los agentes así como en el maestro.

3.2.1 Especificaciónes del maestro

En nuestro caso se decidió ejecutar el maestro en una máquina virtual, utilizando VirtualBox. Se hace uso de una máquina virtual para, en caso de errores de configuración, no dañar el equipo real, y para poder tomar instantáneas de la misma y volver a un estado anterior en caso de corrupción.

En esta máquina virtual se utiliza Ubuntu 18.04 como sistema operativo. Como es necesario acceder a varios puertos desde el exterior, se ha configurado en modo bridge. De esta forma la máquina virtual toma una dirección ip de la misma red que el host real y pudiendo acceder a los servidores que ejecuta de forma cómoda.

Es imprescindible configurarla tambien para utilizar 2 CPUs puesto que son necesarias para Kubeadm y 2GB de memroia RAM.

3.2.2 Maestro del cluster de Kubernetes

El maestro del cluster será el encargado de dar órdenes al resto de dispositivos de la red. Será el que controle todo el cluster, cuándo se ejecutan los Pods, en qué nodos, en qué orden se realiza el rolling-update, etc.

Además, para controlar el cluster será necesario hacer uso del cliente kubectl y que este esté conectado al maestro, ya que el maestro es el que expone la API de kubernetes al resto de internet. Si por ejemplo se intentan realizar acciones sobre el clúster con kubectl directamente sobre un esclavo, se puede comprobar que no es posible, ya que los esclavos reciben órdenes directamente del maestro y no exponen la API de kubernetes, por lo que se obtiene un error de conexión rehusada.

Como el maestro no será un dispositivo IoT, sino una máquina más potente que controla al resto, en vez de usar balena-engine se usará Docker. Como ya se ha comentado, balena-engine se basa en el código fuente de

Docker por lo que son compatibles, utilizaremos Docker para demostrar que es possible crear el cluster con distintos motores de contenedores. Sin embargo, en los esclavos se sigue usando balena-engine porque en ellos sí es necesario optimizar al máximo el espacio ocupado y la carga computacional.

Por motivos de ejecución de ciertos Pods que sirven para controlar el resto de nodos esclavos, la carga computacional del maestro es mayor, se recomiendan 2GB de memoria RAM para que funcione con total fluidez, y, Kubeadm necesitará al menos 2 CPUs.

El proyecto ha sido realizado con estas especificaciones por lo que se puede garantizar que son suficientes.

3.3 Cluster de kubernetes

A continuación, se explca cómo funciona y el proceso de creación del clúster de Kubernetes, que será el medio que se utilizará para distribuir las aplicaciones y actualizaciones a los dispositivos. Además aportará el almacenamiento centralizado de los datos como se comentó.

La herramienta principal para crear el clúster es *Kubeadm*. Kubeadm se apoyará en Kubelet, que es el encargado de ejecutar los componentes en los dispositivos, existirá una instancia de Kubelet por dispositivo que se encargará de gestionar ese dispositivo concreto.

Para el correcto funcionamiento de Kubernetes son necesarios una serie de componentes. Estos componentes en conjunto se denominan plano de control (control-plane). El plano de control está formado por cuatro componentes principales, que se ejecutan **en el maestro** como Pods: etcd, api-server, controler-manager y scheduler.

- Kube-proxy: encargado de hacer llegar las peticiones al Pod correspondiente mediante la manipulación de reglas en el firewall (iptables).
- Etcd: es una base de datos de tipo clave-valor que almacena configuración y datos del clúster. Por ejemplo, las direcciones IP de los Pods y nodos.
- API-server: es la interfaz que expone nuestro clúster para recibir órdenes del exterior. El cliente típico que se utiliza es Kubectl que se conecta al maestro y hace uso de esta API, que es de tipo REST, para crear, eliminar y modificar recursos, donde los recursos representan Pods, Deployments, replica-sets, etc. El api-server es el frontend del plano de control.
- Scheduler: componente del maestro que vigila las órdenes de creación de Pods que llegan a través de la API y se encarga de seleccionar el/los nodo/s que lo ejecutan.
- Controler-manager: este componente se encarga de ejecutar los controladores del clúster. Está compuesto por:
    - Controlador de nodos: es el responsable de notificar y responder cuando un nodo cae.
    - Controlador de replicación: responsable de mantener el número especificado de réplicas de un pod en el sistema.
    - Controlador de endpoints: se encarga de poblar los objetos endpoints, que son principalmente los services y Pods.
    - Controlador de tokens: se encarga de crear tokens de acceso a la API para nuevos espacios de nombres.

Estos componentes podrían ser instalados en el maestro de forma manual como servicios tradicionales, pero se debe asegurar que estos componentes están siempre en ejecución. Para solucionar esto, cuando se ejecuta Kubeadm en el maestro para comenzar a crear nuestro clúster, este automáticamente ejecuta Kubelet, y este a su vez crea una serie de Pods correspondientes a los componentes del plano de control. Al estar Kubelet instalado como un servicio mediante Systemd, éste asegurará que salvo fallos kubelet estará siempre activo y se asegurará de que los componentes del plano de control esten siempre funcionando. Aclarar, que, aunque Kubelet se ejecuta como un servicio en el host, los componentes del plano de control serán Pods (compuestos

por contenedores) que se ejecutarán en el maestro de nuestro clúster. Además, los Pods pertenecientes al plano de control se encontrarán en un espacio de nombres (namespace) especial de Kubernetes llamado *kube-system* y que sirve esencialmente para controlar el clúster y ejecutar Pods que hacen de controladores.

En la siguiente figura se muestran los componentes de un clúster de kubernetes y la forma en que se comunican entre ellos, en nuestro caso, podemos ignorar la conexión con la "Cloud" puesto que no la necesitamos:

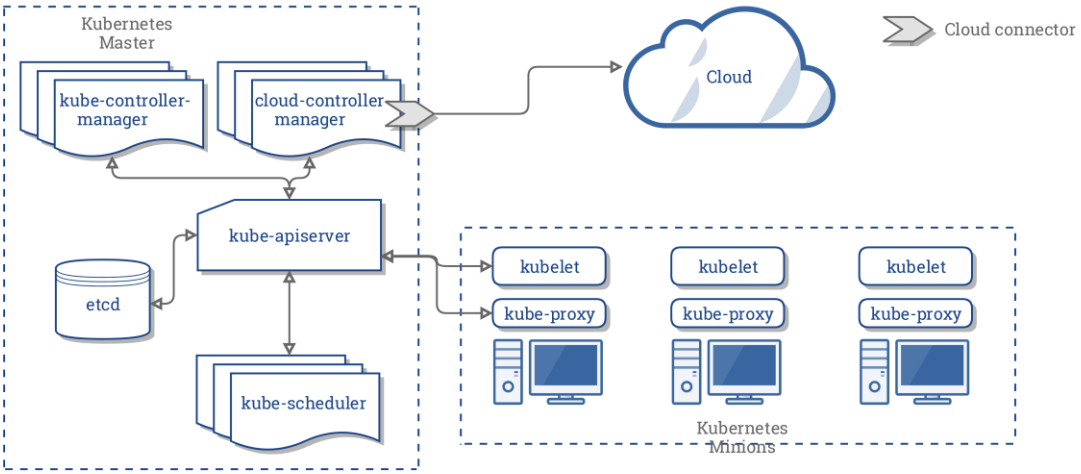

Figura 3-3: Componentes de un clúster de Kubernetes.

El siguiente paso será conectar Kubelet al API-server, es decir, a la API de Kubernetes, lo que permitirá hacer uso del cliente Kubectl para dar órdenes a Kubelet y poder operar sobre los Pods. Este paso lo realiza también automáticamente Kubeadm al iniciar el clúster. Por lo tanto, la API para controlar nuestro clúster de Kubernetes será **expuesta en el maestro**.

En los esclavos, también se necesita tanto Kubeadm como Kubelet. En este caso, Kubeadm se encargará de unirse al clúster que se creó en el maestro, pasandole un token, la dirección ip y puerto del maestro y un resumen SHA. Kubeadm al igual que en el maestro ejecutará Kubelet para gestionar el nodo, pero en este caso no iniciará ningún componente del plano de control, puesto que estos componentes los está ejecutando el maestro.

Todas las instancias de Kubelet (una por nodo), recibirán órdenes del maestro y el maestro las recibirá a través de la API. Aquí cabe destacar que existe una anotación en el maestro por defecto que le dice a Kubernetes que no cree Pods en él. Esto es configurable y puede configurarse para que se ejecuten Pods en el maestro, pero en nuestro caso queremos tener un pod de cada aplicación en cada dispositivo IoT, y se ha hecho que el maestro sea tan sólo una máquina que gestiona los dispositivos, por lo que no queremos que ejecute más contenedores que los del plano de control.

En resumen, se tendrá en cada nodo ejecutándose Kubelet, que controlará los componentes que se ejecutan en el nodo, y Kubeadm, que se encargará de crear el clúster en el maestro y conectarse al clúster en los esclavos.

Al crear el clúster, Kubeadm crea el directorio /etc/kubernetes, este directorio contiene los siguientes archivos y directorios:

- admin.conf: fichero yaml que contiene configuración del clúster. Este fichero será descargado por los esclavos en el momento de la unión al clúster para autoconfigurarse.
- kubelet.conf: fichero yaml que contiene la configuración de kubelet, así como certificados usados para la autenticación con la API.
- manifest: directorio que contiene los ficheros json que definen los componentes (Pods) del plano de control. Kubelet se asegurará de que los Pods definidos dentro de manifest siempre estén ejecutandose.

- pki: contiene los certificacios de la autoridad certificadora (CA), certificados del API-server y tokens de unión al clúster.

Estos ficheros se crean de forma automática, pero podrían ser modificados para cambiar el comportamiento de nuestro clúster.

Si se profundiza sobre el contenido del directorio manifest, y se exploran los ficheros JSON que definen los Pods, se puede comprobar cómo tienen la opción `hostNetwork:true`, lo que indica que estos contenedores estarán disponibles desde fuera del host como si fuesen servidores que no se están ejecutando dentro de contenedores, tomando direcciones de la red en la que se encuentra el host. Recordar que Docker por defecto utiliza redes internas para el resto de contenedores, y estas no son accesibles desde el exterior a menos que se mapee un puerto manualmente.

Existe una limitación en el uso de Kubeadm para la creación del clúster, básicamente no permite, por el momento, la autorreplicación de la base de datos etcd, lo que puede traducirse en que si se cae etcd el clúster quede indisponible.

Otro aspecto importante sobre Kubeadm es que permite la creación de un clúster de un único maestro, se prevé la ampliación en el futuro, pero por el momento solo se puede tener un maestro en el clúster.

3.3.1 Direccionamiento interno del cluster

El último aspecto que debe ser tenido en cuenta en la creación del clúster es la asignación de direcciones a los Pods, ya que cada Pod tiene una dirección única en el clúster. Para ello, cada host ejecutará un Pod especial. Este Pod utilizará un software llamado *Flannel*, que se encarga de gestionar las redes (virtuales) que utilizan los contenedores dentro de un clúster de kubernetes y se encarga de asignar las direcciones a los Pods. Este Pod especial ejecuta un binario llamado *flanneld*. Flanneld se encarga de asignar a cada host un subespacio de direcciones perteneciente a un espacio mayor, que es determinado en el momento de la creación del clúster y pasado como parámetro de Kubeadm (se verá más adelante cuando se expliquen nuestros scripts). Flannel almacena en la base de datos etcd la configuración de la red, subredes asignadas y otros datos auxiliares como las direcciones IP públicas de los hosts. Recordar que etcd se ejecuta en uno de los Pods del plano de control. No debemos confundir este direccionamiento interno del clúster, que es virtual, con el direccionamiento de los dispositivos físicos.

En caso de necesitar comunicación entre los Pods, por ejemplo si se desea que distintas instancias de nuestras aplicaciones que se ejecutan en dispositivos diferentes se comuniquen, existen varios mecanismos para realizar el reenvío de los paquetes entre dispositivos pertenecientes al clúster, llamados “backends” en la documentación de Flannel. Mencionaremos VXLAN, que es un protocolo de superposición de redes, que permite transportar tráfico de capa de enlace sobre capa de red, por ejemplo encapsulando una trama Ethernet sobre UDP. Esto permite que los Pods se comuniquen como si se encontrasen en la misma LAN física, a pesar de que están dentro de dispositivos diferentes. VXLAN es el backend recomendado en la documentación oficial de Flannel. Se muestra una imagen descriptiva del encapsulamiento de las tramas.

Figura 3-4: Encapsulado de trama VXLAN

Existen otras alternativas a Flannel como Cilium, DANM, Contiv y muchas más. En nuestro caso se va a usar Flannel porque es capaz de soportar diferentes arquitecturas (amd y arm entre ellas) y además es más ligero de ejecutar que el resto por lo que lo consideramos el más apropiado para nuestro ámbito de aplicación.

3.3.2 Proceso de creación del clúster

A continuación, se describirá cómo se realiza la creación del clúster, comenzando por el maestro que es el que lo inicia, continuando por los esclavos que se unen a él y terminando por cómo se debe detener el clúster de forma correcta.

Se han automatizado todos estos pasos mediante la creación de scripts en bash, que se verá en el capítulo 4 Escenario de pruebas.

3.3.2.1 Configuración e inicialización del maestro

Lo primero a hacer para crear nuestro clúster será iniciar el maestro. Se enumeran la serie de pasos a realizar:

1. Desactivar el SWAP (se comentó anteriormente el por qué). Para desactivarlo al iniciar el sistema se puede comentar la línea de swap en el ficherto `/etc/fstab`.
2. Instalar Docker.
3. Instalar Kubeadm, Kubelet y Kubectl.
4. Descargar las imágenes de los Pods del plano de control.
5. Iniciar el clúster pasando como parámetro la red base para el direccionamiento interno.
6. Activar el paso por las reglas de iptables de los paquetes que provienen de los contenedores, mediante el parámetro `net.bridge.bridge-nf-call-iptables`.
7. Crear los Pods correspondientes a Flannel, para permitir el direccionamiento interno.

3.3.2.2 Configuración e inicialización de los agentes

I. Instalación de balena-engine

En los agentes en lugar de Docker utiliza balena-engine como se ha comentado en la sección de Tecnologías. La instalación de balena-engine es un poco más compleja que la Docker. También se ha realizado un script que ejecuta los siguientes pasos:

1. Descargar un script secundario de la página oficial de balena-engine.
2. Comprobar la ruta donde deben situarse los ficheros que definen el servicio, `/lib/systemd/system` o `/etc/systemd/system` en función de donde se encuentre el fichero `dbus.service`.
3. Como se está haciendo uso de systemd, descargar el repositorio de GitHub oficial de balena-engine los ficheros `balena-engine.service` y `balena-engine.socket`.
4. Comprobar si existe el grupo balena-engine en el dispositivo y en caso negativo lo crea. Además, añadir al usuario actual al grupo balena-engine. Este paso permite hacer uso de balena-engine sin ser superusuario.
5. Notificar al usuario la siguiente acción a realizar.

La siguiente acción a realizar, que habrá sido notificada por el script es abrir los ficheros `balena-engine.socket` y `balena-engine.service`, que se encontrarán bien en `/lib/systemd/system` o `/etc/systemd/system` según el paso número 2 del script. En dichos ficheros será necesario modificar ciertos parámetros, deben seguirse los comentarios que aparecen en ellos que explican qué debe cambiarse y qué no. Principalmente nos centraremos en descomentar la línea `TasksMax=infinity` de `balena-engine.service` en caso de que la salida del comando `systemctl –version` sea un número mayor de 226.

En el fichero `balena-engine.service` se debe establecer la línea

`ExecStart=/usr/local/bin/balena-engine-daemon`, modificando la ruta que existiese previamente. Por algún motivo la línea original apunta a una ruta que no es la ruta en la que realmente se encuentra el binario de balena-engine. Esto no es un error fácil de detectar ya que simplemente no se ejecutaba el servicio pero no sabíamos el porqué.

Por último, es necesario modificar el fichero `/boot/cmdline.txt` y añadir tres nuevos parámetros: `cgroup_enable=cpuset cgroup_enable=memory cgroup_memory=1`. Estos parámetros se encargan de configurar cgroup para que más tarde, a través del clúster, se puedan ejecutar contenedores en los dispositivos, es decir, permiten a kubelet iniciar los contenedores en el dispositivo. La configuración de estos parámetros no es algo trivial y en caso de no realizarla, cuando se unan los dispositivos al cúster permanecerán como "No preparados" sin arrojar ningún tipo de aviso o error, simplemente no estarán disponibles. Tras configurar estos parámetros es necesario el reinicio del dispositivo para que los cambios se produzcan.

Durante la realización de este proyecto se detectó un bug en la versión de balena-engine que se descarga por defecto el script de la página oficial, ya que esta recibía un error de segmentación cuando se intentaba iniciar. Para solucionar este problema se ha compilado manualmente en otra máquina la versión 17.13.0 de balena-engine que puede encontrarse en Github en la siguiente dirección: https://github.com/balena-os/balena-engine/tree/master.

Una vez descargado el proyecto de Github, para compilarlo debemos ejecutar el script `build.sh`, pero antes, como en nuestro caso lo estamos compilando en una máquina amd64 y lo queremos para una arm, debemos hacer uso de la compilación cruzada de Go (lenguaje en el que está escrito balena-engine). Para ello, antes de ejecutar el script de compilación debemos exportar la variable de entorno `GOARCH=arm`. En el script hay una estructura de control *case,* que en función de esta variable de entorno establece la arquitectura para la que serán compilados los binarios. Una vez compilado, se obtendrá el directorio balena-engine con un ejecutable y una serie de enlaces simbólicos a el. Todos estos ficheros se deben copiar a nuestro agente, en nuestro caso haciendo uso de *scp,* colocarlos en el directorio `/usr/local/bin` (los ficheros sueltos, sin el directorio balena-engine).

Este bug fue corregido más tarde, previo a la finalización del proyecto por lo que el script que se ha creado puede utilizarse para realizar esta instalación de forma automática. En nuestro caso, puesto que se han configurado dos dispositivos, uno ha sido configurado con el script y otro mediante la compilación manual, para asegurar que ambas formas son correctas.

II. Configuración y unión al clúster.

Una vez instalado balena-engine en los agentes, ya tenemos motor de contenedores preparado, necesitamos unir los agentes al clúster y conectarlos con el maestro.

Para ello, debemos seguir los siguientes pasos, algunos similares a los del maestro:

1. Desactivar el SWAP. Para desactivarlo al iniciar el sistema podemos comentar la línea de swap en el ficherto `/etc/fstab`.
2. Instalar Kubeadm, Kubelet y Kubectl de la misma forma que en el maestro.
3. Para permitir a Kubelet, que por defecto estará configurado para usar Docker, utilizar balena-engine, debemos insertar en el fichero `/etc/default/kubelet` la línea `KUBELET_EXTRA_ARGS=--docker-endpoint=unix:///var/run/balena-engine.sock`. Esta línea le indica a kubelet cual es el socket unix utilizado por el motor de contenedores. Este fichero normalmente no existirá, pero basta con crearlo para que kubelet lo lea en su inicio y tome de él los parámetros extra.
4. Establecer el parámetro `net.bridge.bridge-nf-call-iptables=1` .
5. Ejecutar el comando de unión al clúster que aparecerá en el terminal del maestro cuando lo creemos. Este comando posee un token de acceso y un resumen SHA que deben ser correctos para poder unirse al clúster. El comando es de la forma:

```
kubeadm join <master-ip>:<master-port> --token <token> --discovery-token-ca-cert-hash sha256:<hash>
```

Además, necesitaremos pasarle al join la opción `--ignore-preflight-errors=all` ya que Kubeadm busca el comando Docker en el sistema y al no estar presente da un error. Sin embargo, como se ha modificado el socket unix para apuntar a balena-engine, en realidad ese error es inexistente, esta opción hace que la ejecución continue mostrando los errores como avisos y no se detenga en los errores.

Tras estos pasos, para comprobar que se ha unido correctamente, es necesario ir al maestro y ejecutar el comando `kubectl get nodes` que devolverá los nodos existentes en el clúster, el nombre de cada nodo que aparezca será el nombre del host, por ello existía el requisito de Kubeadm de que cada host tuviese un nombre de host diferente.

3.3.2.3 Detención del clúster

Por último, es importante a la hora de detener el clúster hacerlo de la siguiente manera, sino es posible que quede corrupto y no se pueda iniciar de nuevo:

1. Drenar los nodos, incluido el maestro (el último), mediante el comando `kubectl drain <nombre_nodo>`. Esto se encarga de detener todos los Pods del nodo marcarlo para que no se puedan crear nuevos Pods en él.
2. Borrar los nodos con el comando `kubectl delete node <nombre_nodo>`.
3. Reiniciar kubeadm mediante `kubeadm reset` que elimnará todos los archivos del clúster.
4. Por último, eliminar las entradas de iptables que se habrán creado para implementar el cluster: `iptables -F && iptables -t nat -F && iptables -t mangle -F && iptables -X`

Para realizar estos pasos se ha creado también un script que toma como parámetro el nombre del maestro y detiene el clúster de la forma correcta.

3.3.3 Auto-configuración de los esclavos en el proceso de unión

Durante la creación del clúster se produce internamente un proceso de configuración automático de los esclavos que vamos a describir a continuación.

Cuando ejecutamos en el maestro el comando `kubeadm init`, la configuración de kubelet se almacena en el fichero `/var/lib/kubelet/config.yaml` pero, además, se crea un objeto de tipo ConfigMap en el clúster llamado `kubelet-config-1.X` siendo X la versión de kubernetes. El mismo fichero se escribe también en `/etc/kubernetes/kubelet.conf` y será el fichero de configuración de kubelet que utilizarán todos los nodos del clúster. Este fichero de configuración hace referencia a los certificados del cliente que permiten a kubelet comunicarse con el API server. Todo esto permite propagar a los nodos que se van uniendo al clúster la configuración que deben utilizar, como por ejemplo el rango de direcciones IP que habíamos indicado en el momento de la creación del clúster en el maestro.

Además, para poder especificar ciertas características puntuales en ciertos nodos kubeadm almacena un fichero de configuración en `/var/lib/kubelet/kubeadm-flags.env`, que contiene una serie de parámetros (flags) que pasará a kubelet en el momento de su llamada. Es en este fichero donde podemos modificar el Container Runtime utilizado por el nodo. Este fichero podemos modificarlo en cada nodo del clúster de forma independiente.

Al hacer uso del comando `kubeadm join` en los esclavos, kubeadm utiliza el token que le pasamos para leer del clúster el ConfigMap `kubelet-config-1.X` que almacenó el maestro y escribirlo en su propio fichero `/var/lib/kubelet/config.yaml`. Kubeadm crea también el fichero `/etc/kubernetes/bootstrap-kubelet.conf` que contiene el certificado para poder conectarse al clúster y el token anterior. Kubelet utiliza ahora tanto el certificado como el token para realizar una serie de operaciones de autenticación llamadas TLS Bootstrap obteniendo un certificado final que se almacenará en `/etc/kubernetes/kubelet.conf` y será utilizado para la conexión final con el clúster.

3.3.4 Particularidades del sitema

Para este proyecto vamos a necesitar realizar algunas modificaciones sobre el clúster, ya que Kubernetes es una tecnología pensada para el despliegue en la nube y no para el IoT.

El maestro del clúster de forma natural distribuye los Pods según una serie de reglas llamadas *Taints* que se aplican a cada nodo. Cuando queremos realizar un despliegue sobre el clúster y especificamos el número de réplicas de Pods que deben existir mediante el ReplicaSet, estas se distribuyen por los nodos según los Taints hasta alcanzar el número especificado, en algunos nodos se ejecutarán y en otros no. En nuestro caso queremos que las aplicaciones se ejecuten en todos nuestros dispositivos IoT, por lo que el ReplicaSet por defecto no es del todo útil ya que necesitamos que haya al menos un pod ejecutándose en cada dispositivo, si no, tendríamos un dispositivo que no hace nada. Además, podríamos querer especificar los nodos donde se van a crear los Pods, ya que podemos desear ejecutar cierta aplicación en un dispositivo concreto y no en otro.

Para solucionar este problema tan solo debemos sustituir los Deployments por otro objeto de kubernetes llamado *DaemonSet.* Con un DaemonSet conforme añadimos nodos al clúster automáticamente se crean Pods en ellos, y conforme eliminamos nodos del clúster automáticamente los Pods se eliminan. Si eliminamos el DaemonSet todos los Pods se eliminan.

Por defecto un DaemonSet crea Pods en todos los nodos. Para seleccionar solo algunos nodos debemos hacer uso de etiquetas, definiendo en el YAML en el campo `.spec.template.spec.nodeSelector` los pares clave/valor que deben poseer los nodos para que se creen Pods en ellos. Para dar etiquetas a un nodo necesitamos ejecutar el comando `kubectl label nodes <node-name> <label-key>=<label-value>`. Otra opción sería hacer uso de la afinidad de los nodos, pero no la comentaremos puesto que es una característica aun en desarrollo. Un ejemplo de DaemonSet sería el siguiente:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      nodeSelector:
      - app: fluent-logging
      containers:
      - name: fluentd-elasticsearch
        image: k8s.gcr.io/fluentd-elasticsearch:1.20
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

Cuadro de texto 3-2: Ejemplo de definición de un DaemonSet

Un ejemplo de DaemonSet sería el propio Flannel, que hace uso de un DaemonSet para crear los Pods de red que se encargan del direccionamiento interno en cada uno de los nodos del clúster. Este DaemonSet es también multi-arquitectura, y utiliza imágenes diferentes en los Pods que crea en función de la arquitectura del nodo. Por ello, cuando conectamos nuevos dispositivos al clúster, automáticamente se crea en ellos un nuevo Pod encargado del direccionamiento correspondiente al DaemonSet de Flannel. Para lograr esta multiarquitectura el fichero YAML de Flannel (que es el que utilizamos para crear los DaemonSet de direccionamiento al iniciar el maestro) en realidad define varios DaemonSets, uno por cada arquitectura que soporta. De esta manera, solo se crean esos DaemonSet en caso de que exista en el clúster algún dispositivo con esa arquitectura. Por ello, en nuestro clúster tenemos un DaemonSet para dispositivos con arquitectura amd64 y otro para dispositivos con arquitectura arm64.

Otro aspecto a tener en cuenta es que, si queremos tener un clúster que de soporte una red IoT, la cantidad de

dispositivos puede ser muy elevada, lo que implicaría aun mayor cantidad de Pods, por lo que probablemente podríamos pensar en hacer uso de IPv6 en lugar de IPv4. Volvamos a pensar en la naturaleza de las redes internas de nuestro clúster. Recordemos que estas no son accesibles desde el exterior a menos que mapeemos un puerto de un contenedor a un puerto del host y, en cualquier caso, esto nos permite acceso al servicio de ese contenedor y no al resto de contenedores de la red interna. Todos los Pods de nuestro clúster por tanto tendrán direcciones que no son públicas y que por tanto no entrarán en conflicto con otras del resto de internet puesto que nuestra red de Pods es interna y privada. Incluso si mapeamos un puerto, el acceso a ese servicio se realizaría a través de la dirección IP del host y no la del Pod. Por lo tanto, si vamos aumentando el rango de direcciones al crear el clúster cuando lo pasamos como parámetro podemos llegar a tener todas las direcciones posibles de combinar los 32 bits de una dirección IP (excepto las típicas reservadas), es decir, el equivalente a todo internet en direccionamiento. En cualquier caso, aunque no tenemos problemas con las direcciones internas Flannel está trabajando en la implementación de un controlador para IPv6, pero actualmente no es posible su uso. Aclarar que cuando hablamos de un servicio nos referimos a que el dispositivo exponga algún tipo de servidor al exterior, que en muchos casos no es necesario, por ejemplo, si el dispositivo tan solo realiza una medida con un sensor y la almacena. En cualquier caso, como es una posibilidad real que los dispositivos IoT expongan un servicio (evolución de IoT a microservicios) queremos dejar claro este aspecto.

Por otro lado, como se ha comentado, utilizaremos balena-engine en los esclavos y Docker en el maestro. A pesar de ser compatibles en cuanto a CRI, es necesario que en los agentes configuremos el endpoint del servicio de balena-engine para que Kubelet pueda ordenarle a balena-engine crear y eliminar contenedores. Para ello, debemos escribir la línea `KUBELET_EXTRA_ARGS=--docker-endpoint=unix:///var/run/balena-engine.sock` en el fichero `/etc/default/kubelet`. Este fichero será leído por Kubeadm en el momento de ejecutar Kubelet y le indica a Kubelet el socket UNIX donde escucha balena-engine. En el maestro esto no es necesario puesto que por defecto Kubeadm está preparado para funcionar con Docker y dockershim. Puesto que balena-engine es una modificación de Docker no será necesario modificar nada en cuanto al CRI, que por defecto será también dockershim.

También debemos tener en cuenta que la arquitectura de nuestro maestro es amd64, mientras que la arquitectura de los esclavos será arm (ya que se ha escogido Raspberry pi para hacer las pruebas). Hay cierta compatibilidad entre arquitecturas al usar Kubeadm, por lo que en la mayoría de procesos que tienen lugar esto es transparente. Sin embargo, cabe comentar que esto no ocurre con Flannel. Si listamos los Pods exitentes dentro del espacio de nombres kube-system, que es donde se encuentran los Pods de control, veremos cómo existe un pod de Flannel para amd64, que es el que corresponde al maestro y otros Pods para arm que corresponden a los agentes. Sin embargo, al estar los datos del direccionamiento almacenados en la misma base de datos etcd no habrá solapamiento de direcciones.

3.3.5 Despliegue y actualización de aplicaciones

A continuación, vamos a comentar cómo realizar el despligue de una aplicación sobre nuestro clúster.

Para desplegar una imagen en nuestro clúster y que todos los nodos la ejecuten tenemos dos opciones: ir dispositivo por dispositivo construyendo la imagen a partir del Dockerfile con el comando `balena-enfine build -f <fichero-Dockerfile>`, lo cual no es nada practico, o subirla a un registro como DockerHub y que se descarguen en el momento en que se necesiten. Tendremos que tener en cuenta que debemos construir la imagen en una Raspberry ya que si deseamos ejecutarla en un dispositivo con arquitectura ARM deberemos construirla en un dispositivo con arquitectura ARM. Como el maestro no va a ejecutar la imagen, no es necesario disponer de ella para amd64.

Para subir la imagen a DockerHub, debemos ejecutar `balena-engine login` que nos pedirá nuestras credenciales para autenticarnos con el registro, DockerHub por defecto. Una vez autenticados podemos hacer `balena-engine push <organización/imagen>` siendo la organización nuestro nombre de usuario de DockerHub y la imagen se subirá al registro.

Una vez tengamos nuestra imagen en el registro de DockerHub debemos crear el fichero YAML correspondiente al objeto de kubernetes que queramos crear con esa imagen, es decir, puede ser un pod, un Deployment, un ReplicaSet, … En nuestro caso lo más practico es un DaemonSet para que se ejecute en todos

los esclavos, en todos nuestros agentes.

Cuando ya tenemos el fichero correspondiente simplemente tenemos que ejecutar el comando `kubectl create -f <nombre_fichero>` y se realizará el despliegue en todos los esclavos. Todos los esclavos obtendrán la imagen de DockerHub y la almacenarán en su registro local. Este comando debemos ejecutarlo desde un equipo que tenga acceso al API server que expone el maestro, comentaremos más adelante cómo podemos configurar cualquier equipo para esto.

Una vez realizado el despliegue si ejecutamos el comando `kubectl get pods`, suponiendo que se ha utilizado el espacio de nombres por defecto, en otro caso utilizar la opción `-n <espacio_de_nombres>`, obtendremos una lista con todos los Pods que se han desplegado, que si todo ha ido bien debería ser un pod por dispositivo. Estos Pods pueden encontrarse en varios estados dependiendo de la naturaleza de la aplicación (se puede ejecutar de forma permanente, estará en ejecución, o ejecutarse y terminar, estará en terminado) y dependiendo del ciclo de ejecución del pod (inicialización, ejecución, terminado, error, bucle de reinicio, etc).

Como se explicó, cuando un pod finaliza su ejecución, se destruye. Si listamos los Pods en el momento en que la ejecución de todos los contenedores de un pod ha terminado veremos el estado "terminating" y si esperamos un par de segundos y volvemos a listarlos ese pod ya no aparecerá, puesto que se ha destruido.

Pongámonos ahora en el caso de que ya tenemos una imagen distribuida por todos nuestros dispositivos y lo que queremos es actualizarla por cualquier motivo. Gracias a kubernetes no nos vemos obligados a ir dispositivo por dispositivo actualizando las imágenes, sino que simplemente subimos a nuestro registro la nueva imagen con una etiqueta diferente y ejecutamos el siguiente comando `kubectl set image daemonset/<nombre-DaemonSet> <nombre-contenedor>=<nueva_imagen>`.
Otra opción, sería modificar los ficheros YAML que definen el DaemonSet y ejecutar el comando `kubectl apply -f <fichero>` que se encarga de actualizarlo.
Se ha explicado este punto como si lo que teníamos desplegado fuese un DaemonSet, se haría de la misma manera en caso de cualquier otro objeto de kubernetes, aunque para tener la aplicación en todos nuestros agentes necesitamos usar el DaemonSet.

Como vemos, en todo momento estamos controlando el software que se ejecuta en todos nuestros dispositivos como si únicamente tuviésemos uno, puesto que realizamos las acciones en el maestro una sola vez y todos nuestros dispositivos se ven afectados.

Si quisiésemos que se ejecutara algo periódicamente, puesto que los esclavos solo descargan la imagen una vez y la almacenan en su registro local, podemos crear un objeto de kubernetes llamado CronJob, que básicamente da una orden en el momento programado de hacer algo. Ese algo puede ser ejecutar las aplicaciones.

3.3.6 Conexión con el clúster desde el exterior

Por defecto, solo podremos acceder a nuestro clúster y controlarlo con kubectl desde el propio maestro. A continuación, describiremos la manera de comunicarnos con nuestro clúster haciendo uso de equipos externos a él, estos equipos tan solo necesitan el cliente kubectl puesto que no formarán parte del clúster.

Ya que la comunicación con el clúster se realiza a través de una API REST no deberíamos tener la necesidad de acceder al maestro para comunicarnos con el clúster. Si no indicamos lo contrario el cliente kubectl utilizará el fichero de `$HOME/.kube/config` para conectar con un clúster, el que esté ahí indicado. Para solucionar esto, en cualquier equipo que disponga del cliente kubectl podemos establecer un fichero de configuración en otro lugar que apunte a nuestro clúster, es decir, a la dirección del API server, que está expuesto por nuestro maestro, y este fichero debe almacenar también las claves de acceso al clúster, es decir, el certificado. Esto nos permite cambiar rápidamente de clúster a la hora de trabajar con varios, tan solo cambiando una variable de entorno.

Para establecer un nuevo fichero de configuración tan solo debemos exportar la variable de entorno mediante el comando `export KUBECONFIG=/ruta/kubeconfig` y podemos ver qué configuración estamos utilizando mediante el comando `kubectl config view`. En este fichero `kubeconfig` deberá aparecer el certificado para conectarnos con el API server. Este fichero podemos copiarlo directamente del fichero

`$HOME/.kube/config` del maestro. Copiando este fichero completo en cualquier otro equipo y exportando la variable de entorno `KUBECONFIG` para que apunte a él ya podremos conectarnos con nuestro clúster desde un nodo externo a este.

Otra opción es establecer varios clusters en un mismo fichero de configuración, mediante la definición de contextos, pudiendo cambiar entre ellos con el comando `kubectl config use-context`.

Un ejemplo de fichero de configuración para conectarnos al clúster desde cualquier máquina externa se muestra a continuación, este fichero es un fichero real que ha sido utilizado en una de las pruebas realizadas:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN5RENDQWJDZ0F3SUJBZ0l
CQURBTkJna3Foa2lHOXcwQkFRc0ZBREFWTVJNd0VRWURWUVFERXdwcmRXSmwKY2
01bGRHVnpNQjRYRFRFNU1EUXlOekV4TWpZME5Gb1hEVEk1TURReU5ERXhNalkwT
kZvd0ZURVRNQkVHQTFVRQpBeE1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29a
SWh2Y05BUU
    server: https://192.168.100.165:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data:
poK3BtaWpPSFMwdkFiLzBTcS8vNnZCUTJZZU1DV0orOWYzdWliZFQzMVM2TVJCQ
S9sa1ljQ0dOWVZ4cTJrbEprCmE0a2hYUUlEQVFBQm95Y3dKVEFPQmdOVkhROEJB
ZjhFQkFNQ0JhQXdFd1lEVlIwbEJBd3dDZ1lJS3dZQkJRVUgKQXdJd0RRWUpLb1p
JaHZjTkFRRUxCUUFEZ2dFQkFJQzZCV1JWT3RyWmIrSTVKeFJnQmpOUnhmTm9kMU
FjOEN0aQp5bmtNdm94VkZHa1BaMkNUdlM5R0V3OHB6TFRMQWswNXY5M2EwRnNER
2dkWklzbkpCdEFaWDhoYitWU1JXR2RvCjcwL0VnU2Zv
    client-key-data:
01L0kvcFBsNmdPRwpJa0NiazNidDVxN2FEc1NpUjFwa0lOUTJuS0RZNGNVRmk1a
2J4cWFNMlk5blNLK2dRbkQ2MUduSXlicHJrOVB0Cis5L3IrMEFnMGJaUUNoRjl5
KzFPUWl3cXRUS1Q2RXc0cjFoU0RZbGZLZ2k2SXE4Y2l4c3lQb3hJZjRYM0RERmE
KTFRnRHB1K2hBb0dCQU5nVGxuTUtwN3JqbTZpVkRDNTEzbmR6U1ozRDNreUZZYX
FpSUt6azRJVXZvSEQ1d1E5cgo1eDNiZUhBYm95Y2ZKVVhHUVQ4SnBjb1NIeTB5W
```

Cuadro de texto 3-3: Fichero de configuración de Kubectl para acceder a un clúster

Tener en cuenta que este mismo fichero no será válido para ningún otro clúster puesto que en él se especifican los certificados del clúster en el que fue utilizado y no volverán a repetirse para otro clúster.

3.3.7 Visión general sobre la inclusión de dispositivos no Linux

Dado el elevado uso de las placas Arduino se ha considerado que sería intersante indagar acerca de cómo incluirlas en nuestro sistema, los conceptos que presentaremos son extendibles a otras placas de desarrollo que no pueden soportar un kernel Linux. Este sería un buen punto de partida para futuras mejoras de este sistema.

Nos enfrentamos aquí al problema de que el dispositivo debe tener un kernel Linux y en este caso no se cumple. Se ha investigado acerca de si es posible instalar un kernel Linux en un Arduino haciendo uso de

algún módulo externo y la conclusión es que no. Esto se debe a varios factores, vamos a enumerar a continuación los principales:

- La memoria flash de un Arduino, por lo general, es de 32Kb, es posible ampliarla hata 64Kb, pero esto sigue siendo insuficiente ya que un kernel Linux (el más mínimo posible) necesita alrededor de 2MB de memoria para ejecutarse.
- El código fuente de Linux está construido para palabras de 32 bits, mientras que Arduino consta de palabras de 16 bits, excepto algunos modelos como el Arduino101 que es de 32 bits.
- Aunque consiguiéramos realizarlo, Arduino opera por lo general a una frecuencia de reloj de 16MHz por lo que sería demasiado lento.

Aunque las placas Arduino no están diseñadas para soportar esto, existen ciertos modelos de Arduino como el Arduino Yun o el Arduino Tian que ofrecen la posibilidad de tener un kernel Linux, haciendo uso de un microprocesador además del microcontrolador de Arduino en la misma placa. El problema es que estas placas están retiradas del mercado, según la documentación oficial, por lo que no tendría sentido utilizarlas.

Se ha barajado por otra parte la posibilidad de actualizar los *sketch* de forma remota utilizando el protocolo TFTP (*Trivial File Transfer Protocol*). Un sketch no es más que el código que ejecuta Arduino. Esto si que es posible, haciendo uso de módulos de conectividad en las placas Arduino. Sin embargo, tendría ciertas desventajas, como por ejemplo que deberíamos conocer y escribir a mano las direcciones IP de todos los dispositivos por lo que sería complicado de gestionar en caso de tener muchos dispositivos. Por otro lado, también nos obligaría a crear nuevos Pods en el maestro, y estos Pods deberían ir actualizando uno por uno los Arduinos utilizando las direcciones IP que previamente se ha tenido que pasar a mano. No es una solución fácil de gestionar si el número de dispositivos es elevado, pero es una solución posible. En [29] se explica detalladamente y paso a paso como conseguir actualizar un Arduino mediante TFTP.

La solución que nos queda sería hacer uso de otra placa que sí pueda ejecutar un kernel Linux mínimo y que esta a su vez estuviese conectada al Arduino y le actualizase los sketchs (lo más simple sería utilizar el cable serial). Esto se conoce como una *Carrier board* o *placa portadora*. De esta forma, la placa portadora podría unirse al clúster como si fuese otra más, por lo que solucionamos el problema de gestionar las direcciones IP. Tan solo tendríamos que asegurarnos de etiquetar estas placas en el clúster para diferenciarlas del resto, y así poder seleccionar cuáles son nuestros dispositivos de tipo Arduino, puesto que las imágenes deberán portar el Sketch para que la placa portadora lo introduzca en el Arduino, y por tanto, la imagen será diferente para dispositivos como la RaspberryPi.

Para actualizar nuestras placas, debemos construir una imagen que tome de un volumen el sketch y se encargue de introducirlo en el Arduino y debemos distribuir esa imagen a los agentes que tengan la etiqueta que previamente se han debido poner a las placas portadoras.

En la siguiente imagen mostramos un esquema a groso modo de cómo se realizaría esta conexión:

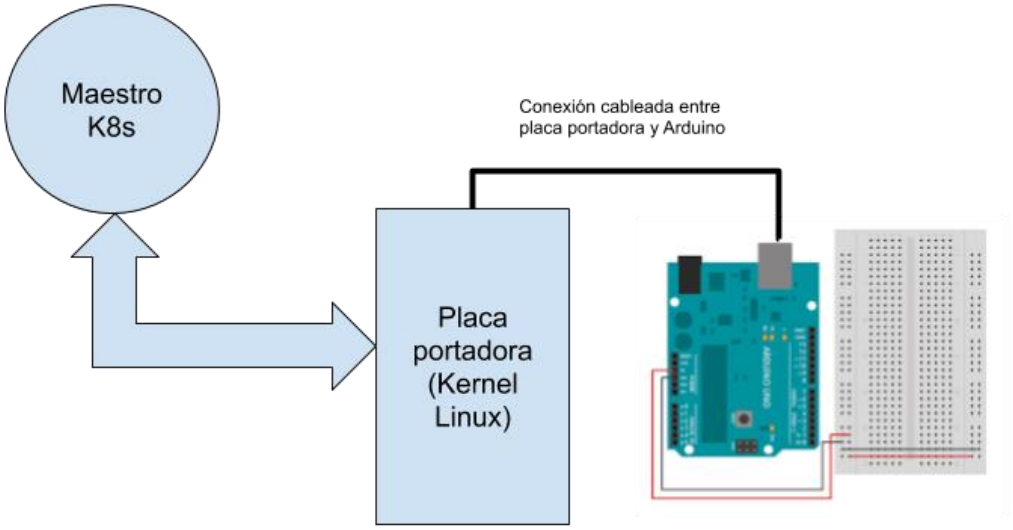

Figura 3-5: Conexión de un Arduino y una placa portadora.

3.4 Creación de imágenes para los dispositivos.

En este apartado vamos a comentar el proceso de creación de una imagen en la que empaquetaremos una aplicación y las librerías de las que depende.

Concretamente vamos a crear tres imágenes diferentes, optimizando significativamente el tamaño de estas respecto a la anterior.

Las aplicaciones que vamos a utilizar hacen uso de sensores de los dispositivos para mostrar cómo desde el contenedor podemos acceder al hardware como se explicó en el apartado 3.1.3 Uso de sensores o hardware desde un contenedor.

Vamos a comenzar por una aplicación que realiza la medida de luz mediante un sensor fotoresistor.

Puesto que no es el objetivo de este trabajo crear las aplicaciones sino empaquetarlas y distribuirlas, vamos a descargar estas aplicaciones de repositorios públicos y de código abierto.

Para crear las imágenes vamos a hacer uso de archivos Dockerfile, como se presentaron en la sección de tecnologías.

Comencemos con el primer Dockerfile, este es el tipo de Dockerfile que crearía un usuario al que no le preocupa el tamaño de la imagen. Como resultado de esta construcción obtenemos una imagen que en su interior tiene un intérprete de python y una distribuición de linux. Como cabe esperar, esta imagen será bastante pesada. El Dockerfile asociado es el siguiente:

```
# Author: Miguel Angel Cabrera Minagorri
# Date: 23/3/2019
# This is the dockerfile to create an image that uses a python program to read the
light level
# with a raspberry pi.

FROM arm32v7/Python

WORKDIR /

RUN apt-get update -y && apt-get install -y git-core sudo --no-install-recommends
&& \
rm -rf /var/lib/apt/lists/*
RUN git clone https://github.com/pimylifeup/Light_Sensor && cp
Light_Sensor/light_sensor.py /
RUN git clone git://git.drogon.net/wiringPi && cd ./wiringPi && ./build
RUN pip install pyserial wiringpi2 RPi.GPIO

ENTRYPOINT ["python"]

CMD ["light_sensor.py"]
```

Cuadro de texto 3-4: Dockerfile de construcción de una aplicación que lee la luz mediante un sensor (sin compilar).

Como podemos ver se ha partido de una imagen base de python (que se basa a su vez en una imagen base de Debian) y se ha clonado el repositorio con la aplicación así como el repositorio con la librería WiringPi para hacer uso de los pines y poder medir el sensor. Como punto de entrada de la imagen se ha establecido el script de python que será interpretado.

Una vez construida la imagen podemos ver que el tamaño de esta es de 760MB, lo que para un dispositivo IoT con poca memoria puede resultar demasiado.

Pasemos a continuación a compilar el código Python gracias a un software llamado PyInstaller. El Dockerfile se muestra a continuación:

```
# Author: Miguel Angel Cabrera Minagorri
# Date: 13/4/2019
# This is the dockerfile to create a binary image to read the light level
# with a raspberry pi.
# To build an image with the minimun size we use a multi stage approach.

####### First stage: prepare and compile the image #######
FROM python AS base
WORKDIR /
RUN apt-get update -y && apt-get install -y git-core sudo --no-install-recommends && \
rm -rf /var/lib/apt/lists/*
RUN git clone https://github.com/pimylifeup/Light_Sensor && cp Light_Sensor/light_sensor.py
/
RUN git clone git://git.drogon.net/wiringPi && cd ./wiringPi && ./build
RUN pip install pyserial wiringpi2 RPi.GPIO pyinstaller
# Compile the python script to a single executable binary that contains all necesary
dependencies.
# This compilation creates the result file into dist directory.
RUN pyinstaller --onedir -y light_sensor.py
############################################################
######## Second stage: copy only the binary of the first stage. #######
FROM scratch
COPY --from=base /dist/light_sensor /light_sensor
COPY --from=base /lib/ld-linux.so.3 /lib/
COPY --from=base /usr/local/lib/libwiringPi.so /usr/local/lib/
COPY --from=base /lib/arm-linux-gnueabi/libm.so.6 /lib/arm-linux-gnueabi/libutil.so.1
/lib/arm-linux-gnueabi/libz.so.1 /lib/arm-linux-gnueabi/libdl.so.2 /lib/arm-linux-
gnueabi/libc.so.6 /lib/arm-linux-gnueabi/libpthread.so.0 /lib/arm-linux-gnueabi/librt.so.1
/lib/arm-linux-gnueabi/libcrypt.so.1 /lib/arm-linux-gnueabi/
ENV PATH="/light_sensor:$PATH"
ENV LD_LIBRARY_PATH="/light_sensor/:/usr/local/lib/:/lib/:/lib/arm-linux-
gnueabi/:$LD_LIBRARY_PATH"
CMD ["light_sensor"]
#####################################################################
```

Cuadro de texto 3-5: Dockerfile de construcción de una imagen que lee la luz mediante un sensor (compilada).

En este caso se ha utilizado una buena práctica llamada *multi stage building* que consiste en construir una imagen partiendo desde imágenes anteriores que realizan todos los procesos necesarios, por ejemplo la compilación, añadiendo a la última imagen (imagen objetivo o target) solo lo absolutamente necesario.

Ahora se ha creado una imagen con tan solo el binario de nuestra aplicación python, por ello, partimos en la imagen objetivo de *scratch,* que no es más que partir desde cero en una imagen. (En realidad no es desde cero

ya que posee ciertas carácterísticas como sistema de ficheros y un directorio raiz).

Vemos que en la imagen objetivo se han copiado también ciertas librerías del sistema. Esto se debe a que PyInstaller no nos permite realizar una compilación estática.

Una compilación estática es aquella en la que todas las librerías se incluyen dentro del binario resultante, por lo que podemos despreocuparnos de si las librerías existen en el sistema en el que se vaya a ejecutar. Por otro lado existe la compilación dinámica que es aquella en las que el binario hace referencia a las librerías mediante su ruta dentro del sistema, por lo que si ejecutamos la aplicación en otro sistema y alguna librería esta en otro lugar no funcionará. La documentación oficial de Docker recomienda hacer uso de una compilación estática para así ahorrarnos el tedioso proceso de copiar las librerías en sus directorios correspondientes.

En nuestro caso, como no es posible hacerlo de forma estática ya que PyInstaller no lo permite, se realiza de forma dinámica y copiamos las librerías a sus respectivos directorios. Para saber qué librerías debemos copiar existe el commando *ldd,* que aplicado sobre un fichero ejecutable nos muestra las librerías de las que depende (y las rutas donde las busca). Como última aclaración las librerías estáticas tienen una extension de archivo ".a" mientras que las dinámicas tienen una extension ".so".

Por último, tras copiar las librerías debemos establecer la variable de entorno *LD_LIBRARY_PATH* que indica al sistema dónde buscar las librerías dinámicas en tiempo de ejecución.

Como resultado de la construcción obtenemos una imagen con un tamaño de 24.7 MB es decir, esta imagen es un 96.75% menor que la anterior.

Por último, se ha creado otra imagen partiendo de un código ligeramente diferente. En este caso la aplicación realiza el control de un semáforo. El código es C y lo compilaremos con gcc. Tampoco podemos compilarlo de forma estática ya que WiringPi no puede compilarse como una librería estática desde marzo de 2018, por lo que igual que antes compilaremos de forma dinámica y copiaremos las librerías. Esta aplicación tiene una lógica ligeramente mayor que la anterior, pero al ser la compilación en C más óptimizada que en Python la imagen será todavía menor. Se muestra el Dockerfile a continuación:

```
# Author: Miguel Angel Cabrera Minagorri
# Date: 13/4/2019
# This is the dockerfile to create a binary image that uses leds on RaspberryPi
# To build an image with the minimun size we use a multi stage approach.

####### First stage: prepare and compile the code #######
FROM gcc AS base

RUN apt-get -y update && apt-get install -y git-core sudo --no-install-recommends && \
rm -rf /var/lib/apt/lists/*
RUN git config --global core.eol lf && git config --global core.autocrlf input
RUN git clone https://github.com/simonprickett/cpitrafficlights.git && \
export PROJECTDIR=`pwd`
WORKDIR /
RUN git clone git://git.drogon.net/wiringPi && \
cd ./wiringPi && ./build
RUN cd $PROJECTDIR/cpitrafficlights/wiringpi && \
make && \
chmod +x trafficlights
############################################################
######## Second stage: copy only the binary of the first stage. #######
FROM scratch
COPY --from=base /cpitrafficlights/wiringpi/trafficlights /
COPY --from=base  /lib/ld-linux.so.3 /lib/
COPY --from=base /usr/local/lib/libwiringPi.so /usr/local/lib/
COPY --from=base  /lib/arm-linux-gnueabi/libpthread.so.0 /lib/arm-linux-
gnueabi/librt.so.1 /lib/arm-linux-gnueabi/libcrypt.so.1 /lib/arm-linux-
gnueabi/libm.so.6 /lib/arm-linux-gnueabi/libc.so.6 /lib/arm-linux-gnueabi/
ENV PATH="/:$PATH"
ENV LD_LIBRARY_PATH="/usr/local/lib/:/lib/:/lib/arm-linux-gnueabi/:$LD_LIBRARY_PATH"
CMD ["trafficlights"]
#####################################################################
```

Cuadro de texto 3-6: Dockerfile de construcción de una imagen para una aplicación que simula un semáforo

Tras crear esta imagen, el tamaño es de 2.39MB es decir es un 99.69% menor que la primera, y además la lógica de la aplicación es más compleja.

Se ha mostrado en este apartado cómo debe enfocarse la creación de las imágenes si queremos que sean lo más óptimas posible en cuanto a tamaño, mostrando progresivamente que se puede ir mejorando una imagen hasta disminuir su tamaño más de un 99%. Por supuesto la disminución del tamaño depende del lenguaje de programación utilizado así como de las librerías necesarias.

Para finalizar este apartado y dado que estas aplicaciones que se han empaquetado las utilizaremos para las pruebas, se va a comentar cómo deben conectarse los pines de la RaspberryPi para cada una de ellas.

Para la aplicación del sensor de luz se debe seguir el siguiente esquema (en nuestro caso hemos cambiado el condensador por una resistencia):

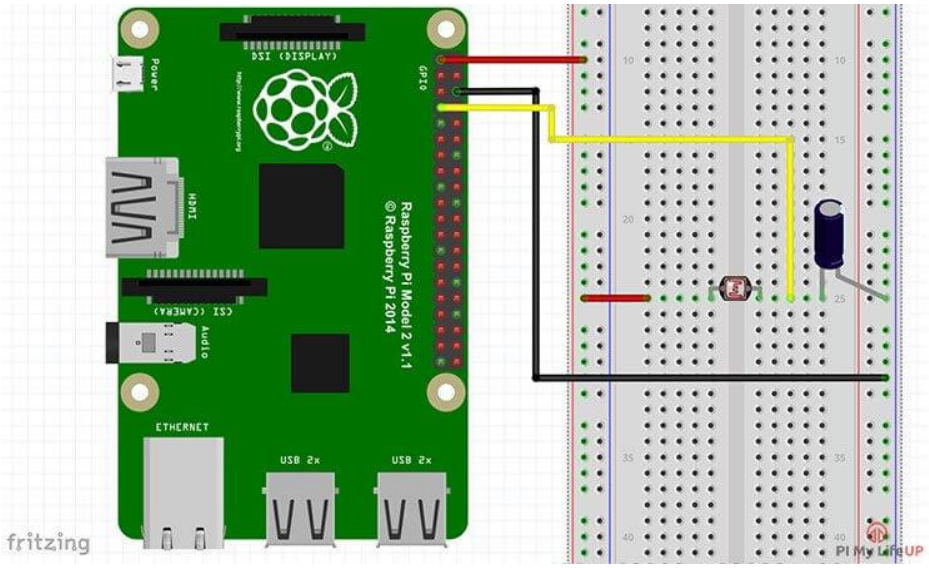

Figura 3-6: Conexión de pines para la ejecución de la aplicación del sensor de luz.

En la imagen anterior vemos cómo debemos conectar la alimentación al pin 1, la tierra al pin 6 y, el punto medio entre el fotorresistor y la resistencia que debe tener en serie. Lo conectamos al pin 7 (en la imagen sale un condensador) que será el punto en el que se tome la medida del voltaje que se transformará a un valor de luminosidad.

Para la aplicación del control de tráfico tan solo debemos conectar los leds amarillo, rojo y verde en los pines 12, 13 y 14 respectivamente. La tierra sigue siendo el pin 6 y la alimentacion el 1. También, deberemos ponerle una resistencia en serie a cada led para evitar quemarlos.

3.5 Resumen del capítulo

A continuación, se va a resumir como quedará finalmente el sistema que se va a desarrollar teniendo en cuenta todo lo anterior.

El clúster de pruebas tendrá tres miembros. Un maestro y dos esclavos. El maestro será una máquina virtual con sistema operativo Ubuntu y de arquitectura amd64. Esta máquina virtual se creará en modo puente para que tome una dirección IP de la red física en la que ese encuentra el host. Por otro lado, los dos esclavos serán un RaspberryPi modelo 3B+ y otra RaspberryPi modelo 3B. Ambas poseen una arquitectura arm64 y utilizan el sistema operativo Raspbian (una distribución de Debian para RaspberryPi).

En cuanto a núcleos de procesamiento el maestro posee 2 (los necesarios para ejecutar Kubernetes) y los esclavos poseen 4 cada uno, aunque con 1 es suficiente.

El maestro tendrá 2GB de memoria RAM y los esclavos 1GB.

En el maestro tendremos desactivada la ejecución de Pods, ya que nuestro objetivo es ejecutarlos en los dispositivos finales. Además, distribuiremos las aplicaciones por medio de objetos Daemonset ya que nuestra intención es que que se creen Pods en todos los nodos que cumplan una condición. Como se verá la condición impuesta es que el nodo posea una etiqueta determinada. Se han etiquetado los nodos como `device=raspberry`. Con la intención de que si en el futuro se uniesen diferentes tipos de dispotivos sea posible controlar qué aplicaciones se ejecutan sobre qué tipo de dispositivos.

Como se ha explicado, en cada nodo se ejecuta Kubelet y Kubeadm, y, en el equipo que haga de cliente, que en este caso será el maestro, pero puede ser cualquier equipo, incluso uno externo al clúster como ya se ha explicado, debe ejecutarse Kubectl (que es el cliente de la API que expone el maestro).

Además de esto en el maestro se ejecuta Docker y en los esclavos balena-engine. En el maestro se ejecuta Docker porque necesitamos que ejecute los Pods del plano de control, cuando se ha mencionado que el maestro no ejecuta Pods nos refererimos a Pods que no pertenezcan al plano de control, es decir de aplicaciones.

A continuación, se muestra de nuevo el esquema con los distintos componentes que se encuentran en cada dispositivo. Los componentes representados dentro de un cuadrado verde corresponden a aquellos que se instalan como un servicio en el propio dispositivo, es decir, como un servicio de sysctl. Los que se encuentran en cuadrados naranjas con las cuatro esquinas redondeadas corresponden a componentes que se ejecutan como Pods. Y el que se encuentra en un cuadrado rojo con dos esquinas cortadas corresponde a un objeto de kubernetes que existe en el sistema y controla los Pods que crea en otros dispositivos. El esquema es el siguiente:

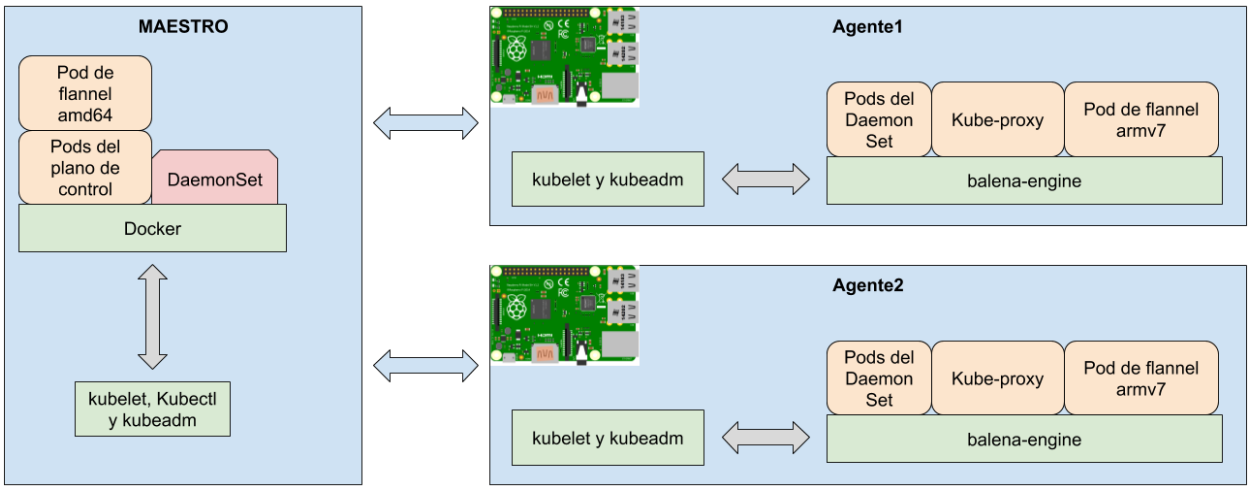

Figura 3-7: Esquema de componentes de los dispositivos en el clúster de pruebas.

En el siguiente capítulo se muestran los scripts que se han desarrollado para la creación del cluster. Se explica lo que hacen, en que orden se deben ejecutar y en que dispositivos.

4 ESCENARIO DE PRUEBAS

En este capitulo se describe el escenario de pruebas utilizado y se presentan los scripts que se han diseñado para crearlo de forma automática. Estos scripts son los que se comentaron también en el capítulo 3 Diseño del sistema. Se detalla a continuación el proceso de creación del clúster, que se comentó brevemente en el capítulo 3 Diseño del sistema.

Lo primero que necesitamos es iniciar el clúster. Esto se hace desde el maestro y para ello se ha creado el siguiente script:

```
#!/bin/bash
# Script to create the master on the cluster.
# Execute whit sudo. Having the sudoers file configured to be root.
# Autor: Miguel Angel Cabrera Minagorri
# Email: devgorri@gmail.com
set -o errexit
set -o nounset
set -o pipefail
# Disable swap, you can comment the swap line on /etc/fstab to avoid swap on booting time
swapoff -a
# Install docker
apt-get install -y \
    apt-transport-https \
    ca-certificates curl \
    gnupg-agent \
    software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
#If amd
add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
apt-get update
apt-get install -y docker-ce docker-ce-cli containerd.io
# Install kubeadm, kubelet and kubectl. This commands are for Debian and Ubuntu
apt-get update -y && apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

```
apt-get update -y
apt-get install -y --allow-change-held-packages kubelet kubeadm kubectl
apt-mark hold kubelet kubeadm kubectl

# Pull control plane images
kubeadm config images pull -v3
#Start the cluster
kubeadm init --token-ttl=0 --pod-network-cidr=10.244.0.0/16

# Deploy flannel network pod
sysctl net.bridge.bridge-nf-call-iptables=1 #On each node of the cluster,
including master

echo ""
echo ""
echo ""
echo ""
echo "Execute the followings commnads a normal user:"
echo ""
echo "
rm -rf \$HOME/.kube/"
echo "     mkdir -p \$HOME/.kube"
echo "     sudo cp -i /etc/kubernetes/admin.conf \$HOME/.kube/config"
echo "     sudo chown \$(id -u):\$(id -g) \$HOME/.kube/config"
echo""
echo""
echo "To deploy the network pod execute:"
echo "   kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/a70459be0084506e4ec919aa1c11
4638878db11b/Documentation/kube-flannel.yml"
```

Cuadro de texto 4-1: master.sh

Este script desactiva el SWAP, instala Docker, Kubeadm, Kubelet y Kubectl en el maestro.

Una vez instalados los componentes necesarios procede a iniciar el clúster utilizando para ello Kubeadm. El primer comando descarga los componentes del plano de control de Kubernetes y el segundo inicia el clúster. Vemos que al de inicio del clúster le pasamos el tiempo de vida del token de unión a cero para que éste no caduque y una dirección base que será la dirección base que utilizarán todos los Pods de flannel (uno por nodo). Esta dirección base se la repartirán entre los Pods de flannel y estos Pods a su vez asignarán direcciones a los contenedores.

Después, modificamos el parámetro del kernel encargado de hacer que todo lo que llega al equipo desde la red pase por el firewall (iptables).

Por último, imprime cuatro comandos que deben ser ejecutados a mano por un usuario no root en el maestro para terminar de configurarlo. Esos comandos básicamente crean el directio `$HOME/.kube` y en el copian la configuración del clúster.

A partir de aquí comienza el proceso que se describió en el capítulo 3 Diseño del sistema en el que la configuración de Kubelet es subida al clúster a modo de Configmap para que los esclavos que se van uniendo la descarguen y la almacenen localmente.

Por último, nos mostrará el comando que debemos ejecutar para crear el Dameonset encargado de crear los Pods de direccionamiento de flannel.

Además de todo esto, el comando de inicio del cúster de Kubeadm nos devolverá el comando que debemos ejecutar en los esclavos para unirlos al clúster, incluyendo el token.

A continuación, debemos instalar balena-engine en los esclavos. Para ello se ha creado el siguiente script:

```
#!/bin/bash
# Shell script to obtain and install the balena daemon.
# Balena daemon is needed to run balena-engine.
# Autor: Miguel Angel Cabrera Minagorri
# Email: devgorri@gmail.com

set -o errexit
set -o nounset
set -o pipefail

#System directory
SYSDIR=/nodir
#temporal file
TMPFILE=install.sh
# Define the group to control the unix socket
GRP=balena-engine

#First of all, download and unpack balena-engine at the correct location (the
unpack is at the downloaded script).
# If this do not work properly we need to compile manually another version
and substitute this point.
curl -sfL -o ${TMPFILE} https://balena.io/engine/install.sh
sed '/cat <<EOF/,/EOF/d' $TMPFILE
chmod +x $TMPFILE
. ./$TMPFILE
rm -f $TMPFILE
```

```
#check the route for the systemd service
if [ -f /lib/systemd/system/dbus.service ]; then
     SYSDIR=/lib/systemd/system
else
     SYSDIR=/etc/systemd/system
fi


#Get the .service and .socket files for systemd service
sudo curl https://raw.githubusercontent.com/balena-os/balena-engine/17.12-
resin/contrib/init/systemd/balena-engine.socket \
  -o $SYSDIR/balena-engine.socket
sudo curl https://raw.githubusercontent.com/balena-os/balena-engine/17.12-
resin/contrib/init/systemd/balena-engine.service \
  -o $SYSDIR/balena-engine.service
# Add the balena-engine group if not exist, and add the actual user to the
balena-engine group
egrep -i "^${GRP}" /etc/group
if [ $? != 0 ]; then

sudo groupadd $GRP
fi
sudo usermod -aG $GRP $USER


#Change needed files. Assuming that the command systemctl --version is
greater thatn 226.
sudo sed -i 's/\#TasksMax=infinity/TasksMax=infinity/g' "$SYSDIR/balena-
engine.service"


#Change the binary route the scripts from the official website are not
correct, the point to another location.
sudo sed -i 's/ExecStart=\/usr\/bin\/balena-engine-daemon -H \
        fd:\/\//ExecStart=\/usr\/local\/bin\/balena-engine-daemon/g' \
        "$SYSDIR/balena-engine.service"
echo ""
echo "Both balena-engine and the daemon have been installed"
echo ""
echo "If you cannot execute balena-engine command without root, logout and
login to re-evaluate the group created"
echo ""
echo "After these changes execute the following commands to start the
service: "
echo "      sudo systemctl daemon-reload"
echo "      sudo systemctl enable balena-engine"
echo "      sudo systemctl start balena-engine"
```

Cuadro de texto 4-2: install_balena.sh

En el capítulo 3 Diseño del sistema ya se ha comentado detalladamente lo que hace este script por lo que no se repetirá aquí.

Una vez instalado balena-engine, se deben preparar los esclavos. Para ello se ha creado el siguiente script:

```
#!/bin/bash
# Shell script to configure the slaves to be ready to join the cluster.
# Execute whit sudo. Having the sudoers file configured to be root.
# Autor: Miguel Angel Cabrera Minagorri
# Email: devgorri@gmail.com
set -o errexit
set -o nounset
set -o pipefail

swapoff -a
sysctl net.bridge.bridge-nf-call-iptables=1

apt-get update -y && apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y --allow-change-held-packages kubelet kubeadm kubectl
apt-mark hold kubelet kubeadm kubectl

#Modify the docker-endpoint, if the file doesn exist create it.
if [[ -f /etc/default/kubelet ]]; then
  sed -i 's/KUBELET_EXTRA_ARGS=.*/KUBELET_EXTRA_ARGS=--docker-endpoint \
  unix:\/\/\/var\/run\/balena-engine.sock/g' /etc/default/kubelet
else
  echo "Creando fichero /etc/default/kubelet"
  sh -c 'sudo echo "KUBELET_EXTRA_ARGS=--docker-endpoint \
    unix:///var/run/balena-engine.sock" > \
    /etc/default/kubelet'
fi

echo "You need to add the following to your /boot/cmdline.txt file and after that
reboot and exec this script again: "
echo "
echo "Once you do that, the slave will be correctly configured and then you can
  join to a cluster executing ´kubeadm join´. The second time you exec this
  script this message will appear again, just do the join to the cluster."
```

Cuadro de texto 4-3: prepare_slave.sh

Este script desactiva el SWAP y activa el parámetro del kernel que hace que todo lo que llega de la red pase por el firewall (iptables).

Una vez hecho esto, instala kubelet, kubeadm y kubectl. Después, modifica (o crea) el fichero `/etc/defatul/kubelet` para cambiar el CRI (Container runtime interface) y apuntarlo al socket Unix de balena-engine.

Por último, imprime un mensaje pidiendo que se modifique el fichero `/boot/cmdline.txt` para añadir los parámetros necesarios y se reinicie el dispositivo. Una vez reiniciado el dispotivo debemos volver a ejecutar el script pero esta vez sin reiniciarlo al final. Y ya tendríamos el dispositivo listo para unirlo al clúster.

Por último, se deben unir los dispositivos al clúster. Para ello copiamos en los esclavos el comando de unión con el token que nos devolvió Kubeadm en el maestro, pero recordamos que es necesario añadir al comando el parámetro `–ignore-preflight-errors=all`. Esto es debido a que se ha cambiado el motor de contenedores y Kubelet va a buscar que Docker esté instalado en el sistema y dará un error si no lo encuentra. Con este argumento le indicamos que no de un error sino que lo muestre como un aviso y continúe su ejecución.

Una vez unido el dispotivo podemos irnos al maestro (que tendrá el cliente de la API REST (kubectl) configurado apuntando a sí mismo, porque es él el que la expone) y ejecutar `kubectl get nodes`. Que nos debe devolver 3 nodos. El maestro más los dos esclavos.

Los nodos se encontrarán en estado NotReady hasta que se creen en ellos los pods de flannel correspondientes.

Podemos ver los Pods del plano de control (incluyendo los de flannel) mediante el comando `kubectl get pods -n kube-system`. Eso nos mostrará los pods del espacio de nombres de control del clúster.

Ahora que nuestro clúster está creado podemos desplegar aplicaciones sobre los dipositivos. Utilizaremos las aplicaciones cuyos Dockerfiles creamos y explicamos en el apartado 3.4 Creación de imágenes para los dispositivos..

Haciendo uso de esas imágenes vamos a crear un objeto Daemonset que aplicará a los nodos etiquetados como `device=raspberry`. Por lo que debemos etiquetar nuestros nodos. Para ello, ejetamos el comando: `kubectl label node <nombre_del_nodo> device=raspberry`. Recordamos que es necesario que los nombres de los hosts esclavos sean diferentes ya que será el nombre del nodo en el clúster.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: my-ds
  namespace: default
  labels:
    k8s-app: my-ds
spec:
  selector:
    matchLabels:
      name: my-ds
  template:
    metadata:
      labels:
        name: my-ds
        device: raspberry
    spec:
      nodeSelector:
        device: raspberry
      containers:
      - name: my-ds
        image: docker.io/miguelaeh/sensorluzcompilado
        tty: true #This is to obtain python logs on stdout because it uses
django
        securityContext:
          privileged: true
        volumeMounts:
          - mountPath: /dev/gpiomem
            name : vol
      volumes:
       - name: vol
         hostPath:
           path: /dev/gpiomem
           type: File
```

Cuadro de texto 4-4: daemonset.yaml

Como vemos en el Daemonset hemos creado dentro del campo *spec* del Replicaset un campo *nodeSelector* que es el que se encarga de que se creen Pods solo en los nodos que tienen la etiqueta *device=raspberry.*

En el campo *image* hemos establecido la ruta a nuestra imagen. Esa imagen es la que hemos creado

previamente y la hemos subido a nuestro repositorio de Dockerhub, para que todos los nodos puedan descargar de ahí la imagen y ejecutarla.

Podemos ver un campo llamada *tty*. Ese campo lo hemos establecido para forzar la salida de los logs de la aplicación a los logs del Pod. Esto por lo general no es necesario cuando la aplicación envía los logs a la salida estándar pero Python es un caso especial ya que utiliza Django para su sistema de logging y necesitamos forzar su impresión.

Por último, vemos cómo hemos realizado el montaje del fichero `/dev/gpiomem` para poder utilizar los pines de nuestra Raspberrypi. Hemos creado un volumen de tipo *File* (fichero) ya que es un fichero que contiene el mapa de memoria de los pines y lo hemos montado en el contenedor en la misma ruta para que la librería GPIO que lleva dentro nuestra imagen sea capaz de localizar el fichero y hacer uso de él para leer y escribir los pines. Obsérvese como es del tipo ReadWriteOnce, lo que indica que solo puede ser leído y escrito por un único nodo del clúster (que será el nodo en el que se encuentra). Aclarar que es posible montar un volumen que exista en otro nodo, pero para nuestro caso no es práctico puesto que queremos leer los pines del nodo en cuestión y no de otro.

El último script que hemos creado sirve para detener el clúster. Este script recibe como parámetro el nombre del nodo master. Vacía todo el contenido del nodo y lo elimina del clúster mediante Kubectl. Al ser el maestro, el clúster se destruye. Por último, resetea la configuración de Kubeadm para que pueda volver a iniciarse con el script de inicio y elimina todas las entradas de las diferentes tablas del firewall.

Es muy importante detener el clúster mediante este script para poder volver a iniciarlo sin problemas una vez detenido ya que, si no lo hacemos así y la configuración de algún componente no se restaura, cuando lo valvamos a intentar iniciar, el maestro quedará corrupto.

El script para detener el clúster es el siguiente:

```
#!/bin/bash
# Script to drain and stop the master of the cluster.
# Execute with sudo. Having the sudoers file configured to be root.
# To works as expected must be given x permissions and be called as
executable not by an interpreter.
# @param master node name.
# Autor: Miguel Angel Cabrera Minagorri
# Email: devgorri@gmail.com
set -o errexit
set -o nounset
set -o pipefail
if [[ $1 ]]; then
  kubectl get nodes
  kubectl drain $1 --delete-local-data --force --ignore-daemonsets
  kubectl delete node $1
  kubeadm reset
  iptables -F && iptables -t nat -F && iptables -t mangle -F && iptables -X
else
  echo "Needed master node name as parameter."
fi
```

Cuadro de texto 4-5: stop.sh

Para finalizar este capitulo, queremos dejar constancia de que los scripts mostrados son válidos sea cual sea el tamaño del clúster o arquitectura de los dispositivos. Los scripts crean un clúster multiarquitectura y la única limitación en cuanto a numero de esclavos será la capacidad de procesamiento del maestro o si el numero de esclavos fuese ridículamente alto (tantos esclavos en el mismo clúster como direcciones IPv4 existentes) de direcciones IP.

Lo único que debe ser adaptado es el Daemonset para las diferentes aplicaciones que se deseen utilizar.

En el siguiente capítulo mostraremos evidencias del funcionamiento de todo lo descrito anteriormente.

5 VALIDACIÓN DE LAS PRUEBAS

Una imagen vale más que mil palabras.

- Anónimo

En este capitulo vamos a demostrar el correcto funcionamiento del sistema propuesto en el capítulo 4 Escenario de pruebas. Haciendo uso de los scripts ya descritos crearemos el clúster, configuraremos y uniremos nuestros nodos y, por útlimo, ejecutaremos como prueba de concepto la aplicación que hace uso del sensor de luz que anteriormente creamos y subimos a nuestro registro de imágenes de DockerHub.

Comencemos haciendo uso del Dockerfile que habíamos escrito para crear nuestra imagen. De los tres Dockefiles que creamos utilizaremos el correspondiente al sensor de luz compilado, ya que el no compilado tarda más en subirse al registro y descargarse de éste. Ejecutando el comando "`balena-engine build -t miguelaeh/sensorluzcompilado -f Dockerfile_compilado .` " se creará la imagen. Ya que la arquitectura destino es arm puesto que será ejecutado en nuestras RaspberryPi´s debemos construirlas en una placa arm también. A continuación, se muestra el final de la salida del comando que indica que se ha creado satisfactoriamente:

```
pi@raspberrypi:~/Light_Sensor $ balena-engine build -t miguelaeh/sensorluzcompilado -f Dockerfile_compilado
 .
Sending build context to Docker daemon  61.44kB
Step 1/15 : FROM python AS base
 ---> 99fa81da368a
Step 2/15 : WORKDIR /
 ---> Using cache
 ---> 8ae0ff9157e7
Step 3/15 : RUN apt-get update -y && apt-get install -y git-core sudo --no-install-recommends &&     rm -rf
/var/lib/apt/lists/*
 ---> Using cache
 ---> 536f177da7d4
Step 4/15 : RUN git clone https://github.com/pimylifeup/Light_Sensor && cp Light_Sensor/light_sensor.py /
 ---> Using cache
 ---> bda85e78263e
Step 5/15 : RUN git clone git://git.drogon.net/wiringPi && cd ./wiringPi && ./build
 ---> Using cache
 ---> 94b1c0c96d50
Step 6/15 : RUN pip install pyserial wiringpi2 RPi.GPIO pyinstaller
 ---> Using cache
 ---> 458a9beb1f3b
Step 7/15 : RUN pyinstaller --onedir -y light_sensor.py
 ---> Using cache
 ---> 8a2db079297c
Step 8/15 : FROM scratch
 --->
Step 9/15 : COPY --from=base /dist/light_sensor /light_sensor
 ---> Using cache
 ---> 8b9920bd0baf
Step 10/15 : COPY --from=base  /lib/ld-linux.so.3 /lib/
 ---> Using cache
 ---> 89efcd8feb83
Step 11/15 : COPY --from=base /usr/local/lib/libwiringPi.so /usr/local/lib/
 ---> Using cache
 ---> 7c1aeab5c1b3
Step 12/15 : COPY --from=base /lib/arm-linux-gnueabi/libm.so.6 /lib/arm-linux-gnueabi/libutil.so.1 /lib/arm-
linux-gnueabi/libz.so.1 /lib/arm-linux-gnueabi/libdl.so.2 /lib/arm-linux-gnueabi/libc.so.6 /lib/arm-linux-gn
ueabi/libpthread.so.0 /lib/arm-linux-gnueabi/librt.so.1 /lib/arm-linux-gnueabi/libcrypt.so.1 /lib/arm-linux-
gnueabi/
 ---> Using cache
 ---> 896980cee49e
Step 13/15 : ENV PATH="/light_sensor:$PATH"
 ---> Using cache
 ---> 3aa451434679
Step 14/15 : ENV LD_LIBRARY_PATH="/light_sensor/:/usr/local/lib/:/lib/:/lib/arm-linux-gnueabi/:$LD_LIBRARY_P
ATH"
 ---> Using cache
 ---> 1252307029d7
Step 15/15 : CMD ["light_sensor"]
 ---> Using cache
 ---> 8cc0821d84ca
Successfully built 8cc0821d84ca
Successfully tagged miguelaeh/sensorluzcompilado:latest
pi@raspberrypi:~/Light_Sensor $
```

Figura 5-1: Construcción de imagen a partir de Dockerfile

En nuestro caso, como ya la habíamos contruido previamente todas las capas de la imagen están en la caché por lo que en vez de crearse de nuevo se toman de ahí.

Una vez tenemos la imagen creada estará almacenada en nuestro registro local, para que los nodos del clúster puedan descargarla necesitamos que esté disponible para ellos desde un registro remoto, por lo que subimos nuestro registro de DockerHub mediante el comando `balena-engine push miguelaeh/sensorluzcompilado` , recordamos que DockerHub es el registro por defecto configurado, pero que puede cambiarse fácilmente. La salida del comando es la siguiente:

```
pi@raspberrypi:~/Light_Sensor $ balena-engine push miguelaeh/sensorluzcompilado
The push refers to repository [docker.io/miguelaeh/sensorluzcompilado]
88fdc75a4bcd: Layer already exists
54ca97637c66: Layer already exists
2b9a9eb38288: Layer already exists
3b930f7eb625: Layer already exists
latest: digest: sha256:e3983a46151caa89b4019dbb313846abdc51debc6fa5758c5eaad502382412d1 size: 1158
pi@raspberrypi:~/Light_Sensor $
```

Figura 5-2: Subir imagen a registro remoto

Como vemos, al subir la imagen al registro remoto se comparan las capas existentes en él con las locales y solo se suben las capas siguientes a la primera que cambie. En nuestro caso, como ya la teníamos nos dice que todas existen por lo que en realidad no sube ninguna. El resumen sha que nos devuelve podemos usarlo para asegurar que estamos utilizando una imagen en concreto, y si cambia algo en la imagen, el resumen no coincidirá y no se descargará, por lo que aporta seguridad ante cambios maliciosos en la imagen.

Una vez tenemos la imagen creada y disponible para ser descargada por los nodos podemos comenzar con la creación del clúster y configuración de nuestros esclavos.

Comencemos por la iniciación del clúster por parte del maestro. Para ello ejecutamos en el maestro (que será nuestra máquina virtual) el script de iniciación `master.sh` (debemos ejecutarlo como superusuario mediante `sudo bash master.sh`). Si todo el proceso termina satisfactoriamente obtenemos la siguiente salida:

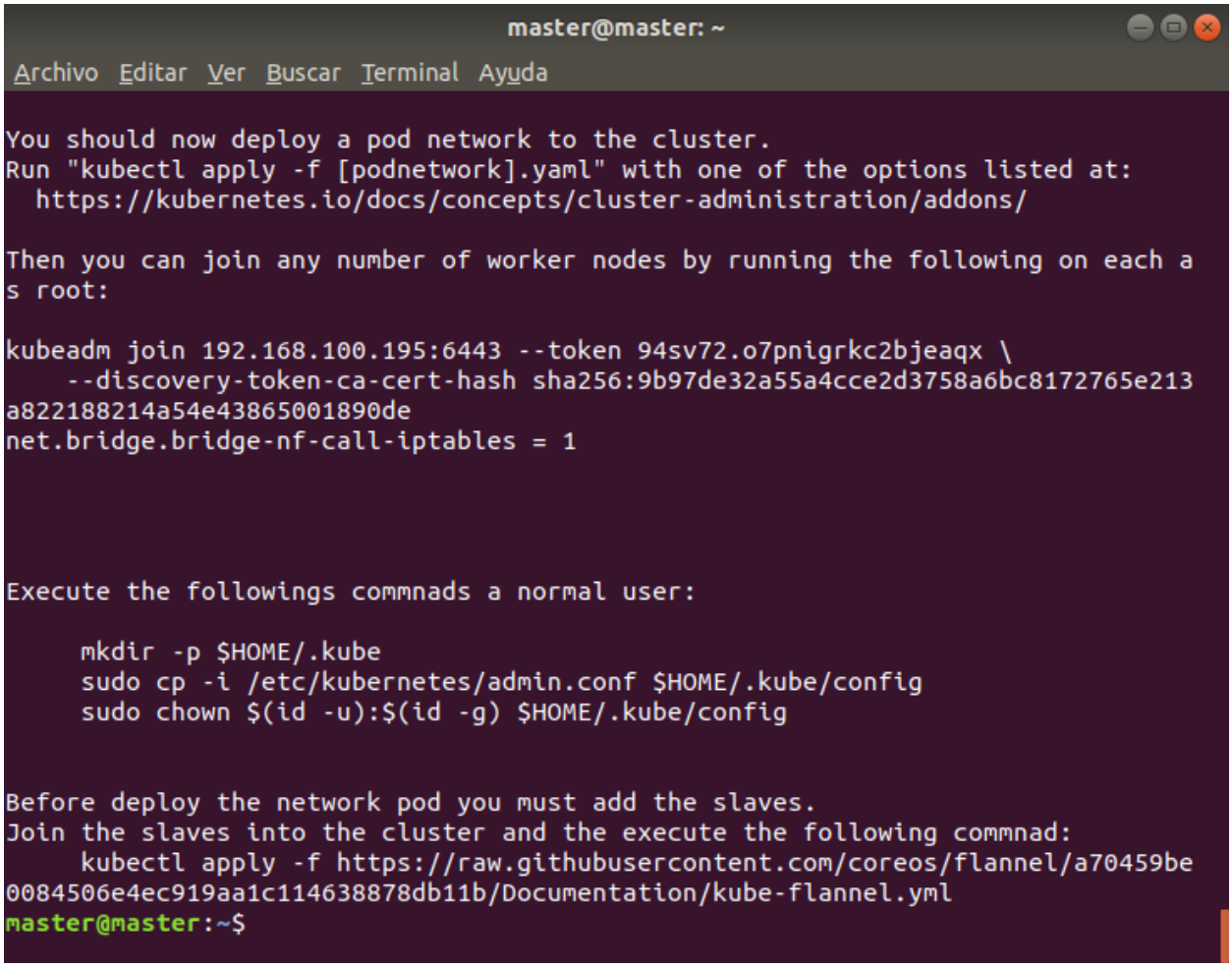

Figura 5-3: Inicio del maestro del clúster

Observamos que en la salida aparece el comando que debemos ejecutar en los esclavos para unirlos al clúster, con el token de acceso y el resumen sha del certificado que el maestro utiliza para Kubelet.

Ahora debemos seguir las instrucciones de la salida del comando anterior para configurar los permisos del directorio de configuración de Kubelet. En nuestro caso eliminaremos primero el directorio para eliminar las configuraciones anteriores que existiesen.

```
master@master:~$ rm -rf ./.kube/
master@master:~$  mkdir -p $HOME/.kube
master@master:~$      sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
master@master:~$      sudo chown $(id -u):$(id -g) $HOME/.kube/config
master@master:~$
```

Figura 5-4: Creación del directorio de configuración de kubelet

Una vez iniciado y configurado el clúster en el maestro podemos desplegar el DaemonSet de Flannel para que conforme añadamos los esclavos se creen sus respectivos Pods de red.

```
master@master:~$ kubectl apply -f https://raw.githubusercontent.com/coreos/flan
nel/a70459be0084506e4ec919aa1c114638878db11b/Documentation/kube-flannel.yml
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.extensions/kube-flannel-ds-amd64 created
daemonset.extensions/kube-flannel-ds-arm64 created
daemonset.extensions/kube-flannel-ds-arm created
daemonset.extensions/kube-flannel-ds-ppc64le created
daemonset.extensions/kube-flannel-ds-s390x created
master@master:~$
```

Figura 5-5: Despliegue del Daemonset de Flannel

Como podemos observar se han creado los pods de flannel correspondientes a cada arquitectura, aunque no serán ejecutados a menos que exista un nodo de esa arquitectura concreta.

Pasamos ahora a la configuración de los esclavos o agentes.

Lo primero que debemos hacer es ejecutar en ellos el script `install_balena.sh` para instalar balena-engine. Para ejecutarlo: `sudo bash install_balena.sh`, obteniendo la siguiente salida:

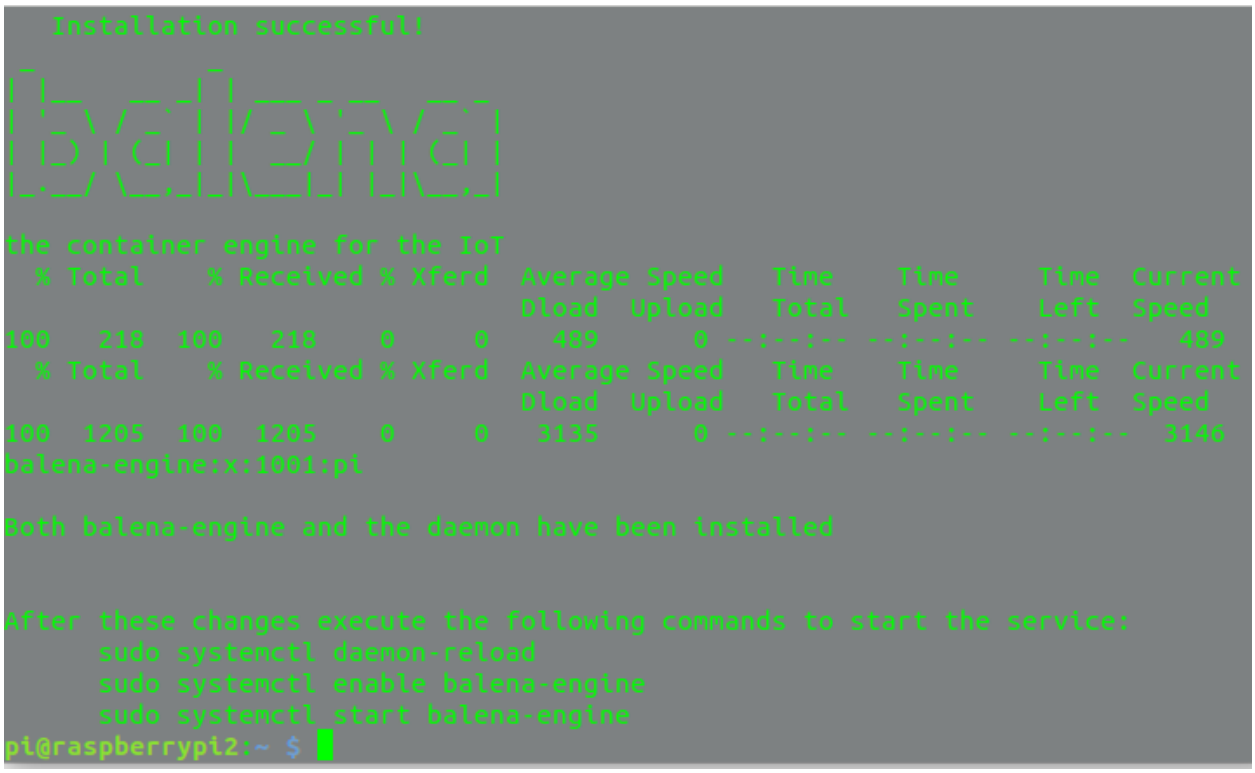

Figura 5-6: Instalación de balena-engine

Debemos iniciar el servicio de balena-engine tal como se dice en la salida anterior.

Una vez instalado balena-engine necesitamos configurar Kubeadm y Kubelet en los esclavos. Para ello ejecutamos el comando `sudo bash prepare_slave.sh` para ejecutar nuestro script de configuración y una vez más realizar manualmente los pasos de la salida, que configuran cgroup en el dispositivo.

La salida del script es la siguiente:

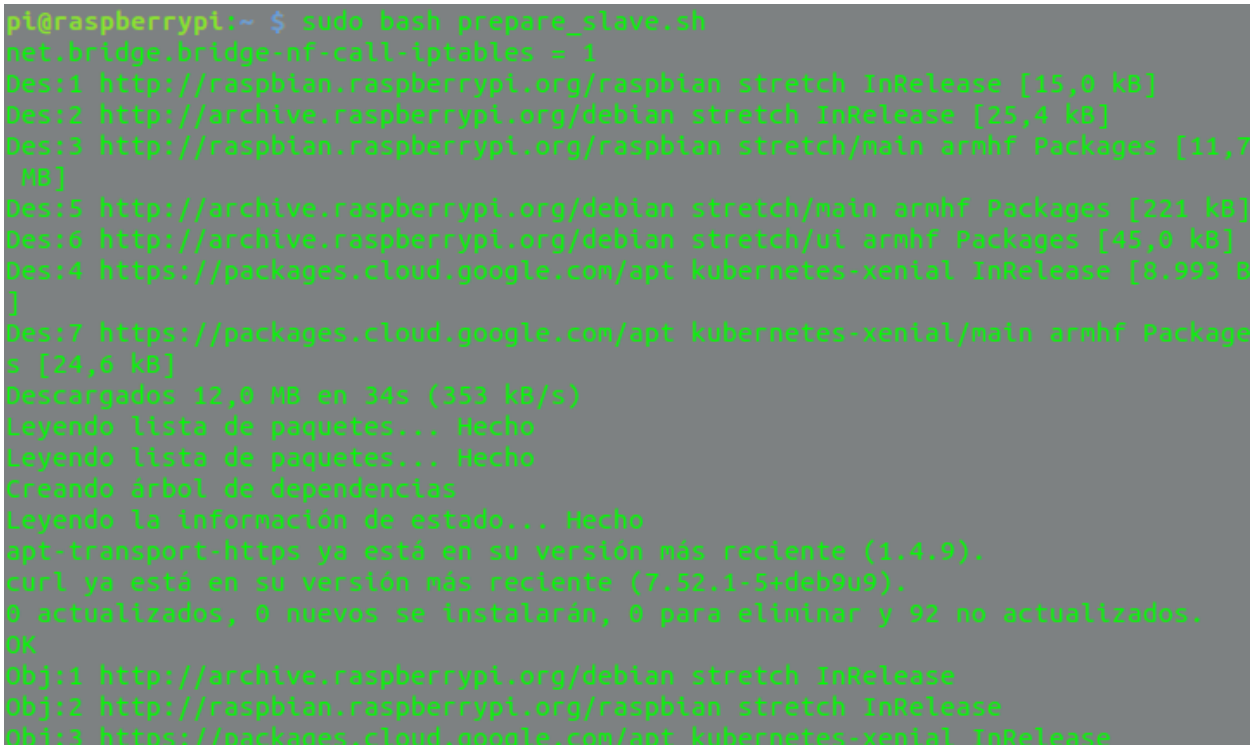

Figura 5-7: Ejecución del script prepare_slave.sh

Y lo que indica que se instalarán (en este caso se actualizarán) los paquetes de Kubeadm, Kubelet y Kubectl: Estos paquetes están retenidos con el fin de que no se actualicen automáticamente en las actualizaciones del sistema ya que podrían hacer que el clúster fallase.

```
Se cambiarán los siguientes paquetes retenidos:
  kubeadm kubectl kubelet
Se actualizarán los siguientes paquetes:
  kubeadm kubectl kubelet
3 actualizados, 0 nuevos se instalarán, 0 para eliminar y 89 no actualizados.
```

Figura 5-8: Salida del script prepare_slave.sh

Una vez que los esclavos están preparados ya podemos unirlos al clúster. El comando a utilizar será el que obtuvimos en la salida del maestro al iniciarlo pero añadiéndole la ya comentada `opción --ignore-preflight-errors=all` para evitar que se detenga cuando no encuentre Docker en el sistema, ya que se ha configurado balena-engine como motor de contenedores y no Docker. Ejecutando el comando:

```
pi@raspberrypi2:~ $ sudo kubeadm join 192.168.100.195:6443 --token mvhqsk.7ghvx1jq
zn8wnp0i --discovery-token-ca-cert-hash sha256:45ea0c90fa7d895df53c6803595baa1a4a0
4c78513e6b8423f1a573f921eea9b --ignore-preflight-errors=all
[preflight] Running pre-flight checks
[preflight] WARNING: Couldn't create the interface used for talking to the contain
er runtime: docker is required for container runtime: exec: "docker": executable f
ile not found in $PATH
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get
 cm kubeadm-config -oyaml'
[kubelet-start] Downloading configuration for the kubelet from the "kubelet-config
-1.14" ConfigMap in the kube-system namespace
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yam
l"
W0526 12:31:42.277440    4938 flags.go:81] cannot automatically assign a '--cgroup
-driver' value when starting the Kubelet: cannot execute 'docker info': executable
 file not found in $PATH
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kube
let/kubeadm-flags.env"
[kubelet-start] Activating the kubelet service
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...
```

Figura 5-9: Unión de esclavo al cluster

Si nos fijamos hay una serie de warnings debido a que se ha cambiado el container-engine, que sin el flag `--ignore-prefligh-errors=all` serían errores y detendrían la ejecución.

El final de la salida que indica éxito, a pesar de los warnings, será el siguiente:

```
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

Figura 5-10: Salida de la unión de un esclavo al clúster

Una vez en este punto, ya tenemos nuestro clúster creado y uno de los dos esclavos unido al mismo. Si obtenemos los nodos del clúster haciendo uso del API-server expuesto por el maestro podemos ver cómo existen dos nodos pero que el esclavo no está listo aún. Esto se debe a que los Pods correspondientes a flannel y a kube-proxy para este nodo se están inciando (estos componentes pertenecen al plano de control, por ello en el comando para listarlos especificamos el espacio de nombre `kube-system`).

En la siguiente imagen vemos que el nodo no está listo y los componentes están en estado de inicio:

```
master@master:~$ kubectl get nodes
NAME           STATUS     ROLES    AGE     VERSION
master         Ready      master   9m19s   v1.14.2
raspberrypi2   NotReady   <none>   37s     v1.14.2
master@master:~$ kubectl get pods -n kube-system
NAME                             READY   STATUS              RESTARTS   AGE
coredns-fb8b8dccf-qk22v          1/1     Running             0          9m3s
coredns-fb8b8dccf-x7gg5          1/1     Running             0          9m3s
etcd-master                      1/1     Running             0          8m6s
kube-apiserver-master            1/1     Running             0          8m6s
kube-controller-manager-master   1/1     Running             0          8m8s
kube-flannel-ds-amd64-j9nmw      1/1     Running             0          4m55s
kube-flannel-ds-arm-gt9fj        0/1     Init:0/1            0          42s
kube-proxy-54whc                 1/1     Running             0          9m3s
kube-proxy-pxs2x                 0/1     ContainerCreating   0          42s
kube-scheduler-master            1/1     Running             0          8m12s
master@master:~$
```

Figura 5-11: Inicio de Pods de control de un nuevo esclavo

Una vez que los componentes del plano de control hayan terminado de inciarse veremos cómo el esclavo estará en estado “Ready” y significará que ya está listo para ejecutar Pods de usuario (los que no pertenecen al plano de control, es decir, de nuestras aplicaciones). Se muestra a continuación:

```
master@master:~$ kubectl get pods -n kube-system
NAME                             READY   STATUS    RESTARTS   AGE
coredns-fb8b8dccf-qk22v          1/1     Running   0          17m
coredns-fb8b8dccf-x7gg5          1/1     Running   0          17m
etcd-master                      1/1     Running   0          16m
kube-apiserver-master            1/1     Running   0          16m
kube-controller-manager-master   1/1     Running   0          16m
kube-flannel-ds-amd64-j9nmw      1/1     Running   0          13m
kube-flannel-ds-arm-gt9fj        1/1     Running   1          9m22s
kube-proxy-54whc                 1/1     Running   0          17m
kube-proxy-pxs2x                 1/1     Running   0          9m22s
kube-scheduler-master            1/1     Running   0          16m
master@master:~$ kubectl get nodes
NAME           STATUS   ROLES    AGE     VERSION
master         Ready    master   18m     v1.14.2
raspberrypi2   Ready    <none>   9m32s   v1.14.2
master@master:~$
```

Figura 5-12: Nuevo esclavo en estado “Ready”

Ahora ya podemos desplegar nuestra aplicación de pruebas, para lo que utilizamos el fichero YAML que define el DaemonSet que se presentó en el capítulo 4 Escenario de pruebas. Una vez lo creemos, si volvemos a listar los pods no existirá ninguno puesto que no se ha puesto una etiqueta al nodo que conincida con el NodeSelector del DaemonSet. Después de poner la etiqueta automáticamente se creará un Pod en el esclavo y podremos verlo al listar los Pods. Se muestran ambos casos en la siguente captura de pantalla, primero se crea el DaemonSet mediante el fichero, listamos los Pods sin haber etiquetado el nodo, etiquetamos el nodo y volvemos a listar los Pods, viendo cómo aparece uno en estado de creación:

```
master@master:~$ kubectl create -f daemonset.yaml
daemonset.apps/my-ds created
master@master:~$ kubectl get pods
No resources found.
master@master:~$ kubectl label node raspberrypi2 device=raspberry
node/raspberrypi2 labeled
master@master:~$ kubectl get pods
NAME          READY   STATUS              RESTARTS   AGE
my-ds-55mfw   0/1     ContainerCreating   0          6s
master@master:~$
```

Figura 5-13: Despliegue de nuestro Daemonset con la aplicación del sensor de luz

Una vez que el Pod se establezca en estado de ejecución “Running” significará que nuestra aplicación está ejecutándose en el nodo de forma correcta, y por lo tanto podremos ver los logs que habíamos configurado para que se enviaran a la salida estándar (recordamos que los enviábamos en el código Python pero además forzábamos la salida en el YAML que define el DaemonSet debido a que el sistema de log de Python lo requiere para poder visualizarlos correctamente).

En la siguiente imagen mostramos como el Pod se encuentra en ejecución y los logs que emite. Estos logs son números que representan cambios en el nivel de luminosidad medido por el sensor, por ello, mientras no cambie aparece un 0 en la medida (hermos pasado la pantalla de un teléfono móvil por encima del fotorresistor para hacer que el nivel de luminosidad cambie):

```
master@master:~$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
my-ds-55mfw   1/1     Running   0          2m29s
master@master:~$ kubectl logs my-ds-55mfw
26089945
0
0
0
0
0
0
5331
0
2391
0
0
```

Figura 5-14: Logs de la aplicación del sensor de luz obtenidos a través del clúster

Se ha demostrado que funciona correctamente con un esclavo, unamos ahora el segundo esclavo para demostrar que funciona con un clúster de varios nodos esclavos.

Siguiendo el mismo procedimiento para unir el nuevo nodo al clúster y esperando a que los componentes del plano de control para el nuevo nodo se creen:

```
master@master:~$ kubectl get nodes
NAME           STATUS   ROLES    AGE   VERSION
master         Ready    master   39m   v1.14.2
raspberrypi    Ready    <none>   26s   v1.14.1
raspberrypi2   Ready    <none>   31m   v1.14.2
master@master:~$ kubectl get pods -n kube-system
NAME                              READY   STATUS              RESTARTS   AGE
coredns-fb8b8dccf-qk22v           1/1     Running             0          39m
coredns-fb8b8dccf-x7gg5           1/1     Running             0          39m
etcd-master                       1/1     Running             0          38m
kube-apiserver-master             1/1     Running             0          38m
kube-controller-manager-master    1/1     Running             0          38m
kube-flannel-ds-amd64-j9nmw       1/1     Running             0          35m
kube-flannel-ds-arm-gt9fj         1/1     Running             1          31m
kube-flannel-ds-arm-kfvq8         1/1     Running             0          37s
kube-proxy-54whc                  1/1     Running             0          39m
kube-proxy-7fr4t                  0/1     ContainerCreating   0          37s
kube-proxy-pxs2x                  1/1     Running             0          31m
kube-scheduler-master             1/1     Running             0          38m
master@master:~$
```

Figura 5-15: Inclusión de un segundo esclavo en el clúster

Ahora aparecen los dos esclavos y el maestro, todos en estado “Ready”. Al igual que pasaba anteriormente, el nuevo nodo no tiene la etiqueta correspondiente para que el DaemonSet cree Pods en él, por lo que inicialmente no los creará. Una vez le pongamos la etiqueta al nodo automáticamente se crearán los nuevos Pods. Puede apreciarse en la siguiente imagen que en cuanto le ponemos la etiqueta aparece un nuevo Pod en estado de creación:

```
master@master:~$ kubectl label node raspberrypi device=raspberry
node/raspberrypi labeled
master@master:~$ kubectl get pods
NAME          READY   STATUS              RESTARTS   AGE
my-ds-55mfw   1/1     Running             0          21m
my-ds-86rcv   0/1     ContainerCreating   0          3s
master@master:~$
```

Figura 5-16: Creación automática de un nuevo Pod en el segundo esclavo

Una vez termine de crearse aparecerán los dos en estado de ejecución:

```
master@master:~$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
my-ds-55mfw   1/1     Running   0          22m
my-ds-86rcv   1/1     Running   0          48s
master@master:~$
```

Figura 5-17: Pods ejecutándose correctamente en nuestros dos esclavos

Esto significa que tenemos dos dispositivos ejecutando un mismo software que ha sido desplegado una única vez (con un único comando). Conforme añadamos y eliminemos nodos se irán creando y destruyendo Pods automáticamente, por lo que, si en algún punto nuestro clúster alcanza, por ejemplo, los mil esclavos habremos desplegado una misma aplicación en mil nodos con un único comando. Además, tendremos el resto de las ventajas que nos aporta el uso de contenedores y que fueron comentadas en capítulos previos.

Para finalizar es necesario detener correctamente el clúster para poder volver a inciarlo en un futuro. Esto lo hacemos mediante el script `stop_master.sh`. Se muestra a continuación la ejecución del mismo:

```
master@master:~$ sudo bash stop_master.sh master
NAME           STATUS   ROLES    AGE   VERSION
master         Ready    master   72m   v1.14.2
raspberrypi    Ready    <none>   32m   v1.14.1
raspberrypi2   Ready    <none>   63m   v1.14.2
node/master cordoned
WARNING: ignoring DaemonSet-managed Pods: kube-system/kube-flannel-ds-amd64-j9n
mw, kube-system/kube-proxy-54whc
evicting pod "coredns-fb8b8dccf-x7gg5"
evicting pod "coredns-fb8b8dccf-qk22v"
pod/coredns-fb8b8dccf-x7gg5 evicted
pod/coredns-fb8b8dccf-qk22v evicted
node/master evicted
node "master" deleted
[reset] Reading configuration from the cluster...
[reset] FYI: You can look at this config file with 'kubectl -n kube-system get
```

Figura 5-18: Detención del clúster mediante el script stop.sh

La salida, que no se muestra completa debido a su longitud, resultará en la detención correcta del clúster y el purgado de los datos del maestro, dejándolo listo para volver a empezar. Cabe destacar que como norma general no se suele detener un clúster ya que el tiempo de vida de estos es bastante elevado, pero en ocasiones es necesario por lo que proporcionamos este script.

6 CONCLUSIONES

Para finalizar esta memoria se evalúa el grado de obtención de los objetivos marcados al inicio de la misma. También se establecen líneas futuras para continuar con el desarrollo del proyecto.

En cuanto a la creación del sistema y la evaluación de la viabilidad del uso de contenedores y Kubernetes para administrar el software en dispositivos IoT, ha quedado demostrado no solo que es posible, sino también que es de gran utilidad. Esto se ve reflejado en el correcto funcionamiento del clúster.

En cuanto a la optimización de los recursos, queda demostrado que es posible combinar en un clúster dispositivos de pocos recursos con máquinas más avanzadas, gracias al uso de motores de contenedores alternativos adaptados a estos. Se considera que, si bien es posible, el esfuerzo que debe realizarse para combinarlos es muy elevado y, al final, la solución parece estar construida a base de parches. Una interesante línea futura sería conseguir que la combinación de diferentes motores de contenedores en un mismo clúster fuese más sencilla.

Atendiendo a las pruebas realizadas, no es posible establecer un mínimo en cuanto a recursos, puesto que no se dispone de dispositivos variados con los que hacer pruebas. Sin embargo, es posible establecer un umbral inferior para los dispositivos IoT con el que se asegura el correcto funcionamiento. Este umbral inferior viene impuesto por las pruebas realizadas y es el siguiente:

- Kernel Linux.
- 1 GB de RAM.
- 1 CPUs (o núcelo de procesador). En nuestro clúster tienen 4 pero es suficiente con 1 en los esclavos.

Un aspecto importante en cuanto al uso de recursos es la batería. En todo momento se han utilizado los dipositivos conectados a la red eléctrica por lo que no han surgido problemas con la batería. Sin embargo, debería estudiarse en qué medida podría afectar a la duración de la batería el mantener una conexión continua con el clúster. Sin duda este sería un aspecto interesante como objeto de investigación en el futuro. Otro aspecto interesante relacionado con este sería indagar acerca del uso de otras tecnologías de comunicación como LoRa o ZegBee en lugar de Wifi ya que son tecnologías más centradas en la optimización de la batería.

Como se comentó al principio de esta memoria, tradicionalmente estas tecnologías se utilizan para procesar datos que envían las flotas de dispositivos. En este caso, se ha dado un enfoque diferente y se ha gestionado directamente el software de los dispositivos, quedando demostrado que es posible aplicar estas tecnologías a otras áreas, no solo a la computación en la nube, con pequeñas adaptaciones.

Por último, se han creado dos aplicaciones como prueba de concepto para demostrar que toda la teoría comentada puede llevarse a la práctica. Estas aplicaciones han permitido llegar a la conclusión de que las adaptaciones realizadas funcionan correctamente y por lo tanto se puede considerar exitosa la adaptación al IoT de estas tecnologías.

REFERENCIAS

[1]"What is Kubernetes", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed: 25- May- 2019].

[2]"The Kubernetes API", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/concepts/overview/kubernetes-api/. [Accessed: 25- May- 2019].

[3]"Kubernetes Components", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed: 25- May- 2019].

[4]"Taints and Tolerations", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/. [Accessed: 25- May- 2019].

[5]"DaemonSet", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. [Accessed: 25- May- 2019].

[6]"balenaOS - Home", *balenaOS*, 2019. [Online]. Available: https://www.balena.io/os/. [Accessed: 25- May- 2019].

[7]"Explaining OverlayFS – What it Does and How it Works", *Datalight.com*, 2016. [Online]. Available: https://www.datalight.com/blog/2016/01/27/explaining-overlayfs-%E2%80%93-what-it-does-and-how-it-works/. [Accessed: 25- May- 2019].

[8]I. Lewis, "How kubeadm Initializes Your Kubernetes Master - Ian Lewis", *Ianlewis.org*, 2018. [Online]. Available: https://www.ianlewis.org/en/how-kubeadm-initializes-your-kubernetes-master. [Accessed: 25- May- 2019].

[9]"balenaEngine - Home", *balenaEngine*, 2019. [Online]. Available: https://www.balena.io/engine/. [Accessed: 25- May- 2019].

[10]"balena-os/balena-engine", *GitHub*, 2019. [Online]. Available: https://github.com/balena-os/balena-engine. [Accessed: 25- May- 2019].

[11]"Moby", *Mobyproject.org*, 2019. [Online]. Available: https://mobyproject.org/. [Accessed: 25- May- 2019].

[12]"containerd", *Containerd.io*, 2019. [Online]. Available: https://containerd.io/. [Accessed: 25- May- 2019].

[13]"containerd/containerd", *GitHub*, 2019. [Online]. Available: https://github.com/containerd/containerd. [Accessed: 25- May- 2019].

[14]"opencontainers/runc", *GitHub*, 2019. [Online]. Available: https://github.com/opencontainers/runc. [Accessed: 25- May- 2019].

[15]"opencontainers/runtime-spec", *GitHub*, 2019. [Online]. Available: https://github.com/opencontainers/runtime-spec. [Accessed: 25- May- 2019].

[16]"Docs Overview – Yocto Project", *Yoctoproject.org*, 2019. [Online]. Available: https://www.yoctoproject.org/docs/. [Accessed: 25- May- 2019].

[17]"Post-installation steps for Linux", *Docker Documentation*, 2019. [Online]. Available: https://docs.docker.com/install/linux/linux-postinstall/. [Accessed: 25- May- 2019].

[18]"Home - Zephyr Project", *Zephyr Project*, 2019. [Online]. Available: https://www.zephyrproject.org/. [Accessed: 25- May- 2019].

[19]"Introduction — Zephyr Project Documentation", *Docs.zephyrproject.org*, 2019. [Online]. Available: https://docs.zephyrproject.org/latest/introduction/index.html. [Accessed: 25- May- 2019].

[20]"Enterprise Application Container Platform | Docker", *Docker*, 2019. [Online]. Available: https://www.docker.com/. [Accessed: 25- May- 2019].

[21]"Docker Documentation", *Docker Documentation*, 2019. [Online]. Available: https://docs.docker.com/. [Accessed: 25- May- 2019].

[22]"Installing kubeadm", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/setup/independent/install-kubeadm/. [Accessed: 25- May- 2019].

[23]"Creating a single master cluster with kubeadm", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/. [Accessed: 25- May- 2019].

[24]"Configuring each kubelet in your cluster using kubeadm", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/setup/independent/kubelet-integration/. [Accessed: 25- May- 2019].

[25]"Configuring each kubelet in your cluster using kubeadm", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/setup/independent/kubelet-integration/#the-kubelet-drop-in-file-for-systemd. [Accessed: 25- May- 2019].

[26]"Organizing Cluster Access Using kubeconfig Files", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/. [Accessed: 25- May- 2019].

[27]"Light_Sensor", *GitHub*, 2019. [Online]. Available: https://github.com/pimylifeup/Light_Sensor. [Accessed: 25- May- 2019].

[28]"simonprickett/cpitrafficlights", *GitHub*, 2019. [Online]. Available: https://github.com/simonprickett/cpitrafficlights.git. [Accessed: 25- May- 2019].

[29]"Arduino Playground - TFTPBootloader1", *Playground.arduino.cc*, 2018. [Online]. Available: https://playground.arduino.cc/Code/TFTPBootloader1/. [Accessed: 01- Jun- 2019].

[30]"Volumes", *Kubernetes.io*, 2019. [Online]. Available: https://kubernetes.io/docs/concepts/storage/volumes/. [Accessed: 01- Jun- 2019].