Universidad de Castilla-La Mancha

Escuela Superior de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Tecnología Específica de

INGENIERÍA DE COMPUTADORES

EXPLORANDO LAS CAPACIDADES DEL PLANIFICADOR DE KUBERNETES EN ENTORNOS DE COMPUTACIÓN EN LA NIEBLA

Ángel Ortega Alfaro

Julio, 2021

Trabajo Fin de Grado

Grado en Ingeniería Informática

Tecnología Específica de

INGENIERÍA DE COMPUTADORES

EXPLORANDO LAS CAPACIDADES DEL PLANIFICADOR DE KUBERNETES EN ENTORNOS DE COMPUTACIÓN EN LA NIEBLA

Autor: Ángel Ortega Alfaro

Tutoras: Mª Blanca Caminero Herráez

Carmen Carrión Espinosa

Julio, 2021

Para mi abuelo, que siempre está presente.

Declaración de Autoría

Yo, Ángel Ortega Alfaro con DNI, declaro que soy el único autor del trabajo fin de grado titulado “Explorando las capacidades del planificador de Kubernetes en entornos de computación en la niebla” y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual y que todo el material no original contenido en dicho trabajo está apropiadamente atribuido a sus legítimos autores.

Albacete, a 13 de Julio de 2021.

Fdo: Ángel

Resumen

El crecimiento del Internet de las Cosas está suponiendo que las arquitecturas Cloud centralizadas no puedan soportar todos los requisitos de los servicios de IoT, como problemas con el tráfico de red y la dificultad de no tener latencias lo suficientemente bajas. En respuesta a esta problemática, surge la Computación en la Niebla (*Fog Computing*). Este nuevo tipo de arquitectura de red busca reducir el caudal de datos de tal forma que los nodos más externos de la red puedan procesar datos localmente, aligerando así la carga de red.

Por otro lado, las tecnologías de virtualización y orquestación de contenedores encajan perfectamente en este tipo de arquitecturas, ya que las características de los contenedores permiten una gran flexibilidad y seguridad en su uso. Kubernetes es el orquestador de contenedores más usado en la actualidad, siendo su planificador el eje central de este TFG. La función principal del planificador de Kubernetes es desplegar los contenedores en los distintos nodos del cluster de Kubernetes, siendo el objetivo de este TFG realizar una evaluación sobre sus distintas funcionalidades.

Esta evaluación se ha realizado sobre computadores monoplaca, concretamente con varias *Raspberry PI,* ya que son unos dispositivos con una estupenda relación calidad/precio y muchas arquitecturas de computación en la nube reales usan estos ordenadores monoplaca como nodos de red. La experimentación sobre distintos escenarios nos ha permitido conocer de primera mano cómo funciona el planificador y la importancia de conocer cuantos recursos consume cada contenedor desplegado.

Agradecimientos

En primer lugar me gustaría agradecer a toda mi familia el apoyo durante estos años ya que sin ellos no hubiera podido llegar hasta hoy.

También me gustaría agradecer a mis tutoras Blanca y Carmen, que solo tengo buenas palabras para ellas, su apoyo y dedicación en todo el transcurso del desarrollo de este trabajo, que no habría sido posible sacar adelante sin su ayuda.

Por ultimo también me gustaría agradecer tanto a los compañeros que he conocido en la universidad, que han pasado a ser verdaderos amigos, como a los que siempre han estado ahí el poder contar siempre con ellos tanto para los malos como para los buenos momentos.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En este capítulo, se desarrollará el contexto sobre el que se ha construido este TFG. Además, se expondrán los objetivos perseguidos y la metodología utilizada en la elaboración del proyecto.

1.1 Introducción

Mientras que el Internet de las cosas (IoT) se está expandiendo a pasos agigantados, las arquitecturas Cloud centralizadas no pueden sostener todos los requisitos necesitados por los servicios de IoT en ciertos casos como las *Smart Cities* [1]. Las grandes demandas de tráfico y los problemas relativos a la dificultad de no tener latencias lo suficientemente bajas convierten estas arquitecturas Cloud en soluciones poco prácticas. Como respuesta, la computación en la niebla (*Fog computing*) surge para dar respuesta a esta problemática. El objetivo de la computación en la niebla consiste en acortar las vías de comunicación entre la nube y los dispositivos, de tal manera que se reduzca el caudal de datos en redes externas. Así, los nodos cumplirían un papel de capa intermedia en la red que se decide qué datos se procesan localmente y cuáles se envían a la nube para ser procesados [2].

Por otro lado, Kubernetes es una tecnología basada en contenedores ampliamente usada en este tipo de arquitecturas *Cloud* y *Fog*, dando la posibilidad de administrar y automatizar las operaciones de contenedores [12]. Por lo general, cuando tenemos un

sistema basado en Kubernetes, tenemos un clúster con un nodo maestro que se encarga de la gestión de dicho clúster, y un conjunto de nodos esclavos en donde se ejecutan todos estos contenedores. Uno de los componentes que usa el clúster de Kubernetes para gestionar y mapear cada despliegue de contenedores en uno u otro nodo es el planificador [16].

Así nace la motivación de este TFG, en el que se pretende realizar un estudio sobre el comportamiento del planificador por defecto de Kubernetes y cuáles son los métodos que pueden influir en su comportamiento, así como la posibilidad de incluir otros planificadores que permitan particularizar la planificación de ciertos despliegues.

1.2 Objetivos

Dicho lo anterior, este TFG está motivado por el hecho de conocer el funcionamiento del planificador de Kubernetes, de tal forma que se puedan optimizar la ejecución de los contenedores desplegados en el cluster según sus necesidades.. De este modo, en este TFG se estudian y valoran las distintas opciones de planificación que tiene el usuario del clúster, de tal forma que, cuando se tenga una arquitectura de computación en la niebla real, se puedan optimizar las aplicaciones que se ejecutan sobre ella.

Por tanto, los objetivos de este TFG son:

i) Estudio del funcionamiento de la tecnología Kubernetes basada en contenedores.

ii) Estudio del funcionamiento de la placa Raspberry Pi, ya que será usada a modo de nodo *fog*.

iii) Familiarización con la plataforma Kubernetes sobre las Raspberry Pi.

iv) Estudio del funcionamiento del planificador de Kubernetes y todas sus posibilidades.

v) Implementación y evaluación de distintas estrategias de planificación y distintos escenarios de estrés para comprobar el rendimiento del planificador en este tipo de situaciones.

La realización de este TFG supone la obtención de 12 créditos ETCS. Puesto que cada crédito corresponde a, aproximadamente, unas 25 horas de trabajo, la realización de

este TFG debe estar comprendida en unas 300 horas. Por tanto, vamos a realizar una estimación del tiempo consumido por cada una de las fases de este proyecto:

- Instalación y configuración de las Raspberry Pi y el software necesario para poner en marcha el cluster de Kubernetes: **50h**
- Estudio de Kubernetes y las opciones de su planificador: **100h**
- Evaluación de rendimiento del planificador y otras pruebas: **100h**
- Redacción de la memoria del TFG: **40h**

Cabe indicar que ésta planificador es orientativa, ya que algunas de las fases se han solapado entre ellas, como las últimas pruebas de evaluación las cuales se han ido realizando a la vez que se escribía esta memoria.

Durante el desarrollo de este TFG se ha hecho uso de una metodología de desarrollo ágil basada en SCRUM [55], en donde se han mantenido reuniones quincenales con las tutoras para un correcto seguimiento del trabajo. Todas estas reuniones han sido realizadas en su mayoría a través de la Plataforma corporativa Microsoft Teams [54], que resultó de gran utilidad.

1.3 Estructura del proyecto y la memoria

La memoria de este TFG está estructurada de la siguiente manera:

i) En primer lugar, en el capítulo actual se presenta el contexto, los objetivos del TFG, y la estructura de este documento.

ii) A continuación, en el capítulo 2 se realizará un repaso de los principales conceptos teóricos tratados a lo largo de todo el TFG, realizando una visión más profunda en lo relativo a Kubernetes y su planificador.

iii) Después, en el capítulo 3, se explicará la arquitectura planteada, indicando el proceso de implementación del cluster, las pruebas que han sido realizadas, y qué metodología se ha usado para analizar los resultados obtenidos.

iv) A continuación, en el capítulo 4, se presentarán los distintos resultados obtenidos.

v) Por último, en el capítulo 5, se detallarán las conclusiones a las que se han llegado tras la realización de este TFG. En este apartado también se indicaran cuáles podrían ser las líneas de trabajo que podrían surgir a partir de los

resultados de este proyecto, aportando algunas indicaciones a seguir en futuros trabajos o investigaciones. Para finalizar, se indicarán las competencias obtenidas por el autor durante la realización de este TFG.

Cabe indicar que en los anexos del TFG se recogen el código y los scripts de Linux que se mencionan a lo largo de la memoria.

Capítulo 2

Estado del Arte

La computación en la niebla es una arquitectura descentralizada cuyo objetivo es minimizar la latencia en aplicaciones IoT y reducir los requisitos de ancho de banda sobre las infraestructuras Cloud, servicios de computación y red físicamente situados más cerca de los usuarios finales. Generalmente, estas arquitecturas consisten en una red formada por nodos "*fog*" los cuales ejecutan microservicios sobre contenedores. Una posible configuración de esta arquitectura sería un conjunto de Raspberrys a modo de nodo, conectadas a través de un clúster de Kubernetes [24].

En el desarrollo de este capítulo se presenta una visión general de los conceptos tratados durante el desarrollo del trabajo, profundizando en aquellos que son el eje central del mismo.

2.1 Computación en la niebla

Debido a algunas de las características del IoT, esta tecnología presenta algunos límites de computación en términos de energía y almacenamiento, además de contar con muchos problemas de rendimiento, seguridad, privacidad y fiabilidad. La integración del IoT con la nube, conocido como *Cloud of Things* (CoT), es la forma correcta de solventar la mayoría de estos problemas. El CoT simplifica el flujo de datos del IoT reuniéndolos y procesándolos, proveyendo de una rápida instalación e integración de bajo coste para el procesamiento de datos complejos y el despliegue de aplicaciones [1].

La integración del IoT con la computación en la nube ha traído muchas ventajas a un gran conjunto de aplicaciones de IoT. Sin embargo, puesto que hay una gran cantidad de dispositivos IoT dispuestos en plataformas muy heterogéneas entre ellas, el desarrollo de aplicaciones para IoT es una tarea compleja. Esto es debido a que las aplicaciones IoT generan una gran cantidad de datos obtenida de sensores y otros dispositivos. Estos datos son analizados para determinar distintos tipos de situaciones, como la gestión de tráfico en *Smart Cities*, en dónde la toma de datos en tiempo real permitiría controlar dispositivos de regulación del tráfico, de tal forma que la circulación de vehículos sea más fluida [25] o la gestión del agua, en dónde desde que se canaliza en los puntos de recogida, su tratamiento, distribución es necesaria toda una red de sensores que es capaz de monitorizar cada uno de estos procesos, con la mayor precisión posible y con el mayor control del que seamos capaces [26]. Para poder mandar toda esta cantidad de datos, a la nube, se requiere un ancho de banda muy alto, por lo que, para solucionar estos problemas, la computación en la niebla entra en juego. Se define computación en la niebla como un paradigma con capacidades limitadas con servicios de computación, almacenamiento y conexión en red de forma distribuida entre diferentes dispositivos finales y la clásica computación en nube. Proporciona una buena solución para las aplicaciones de IoT que son sensibles a la latencia [1].

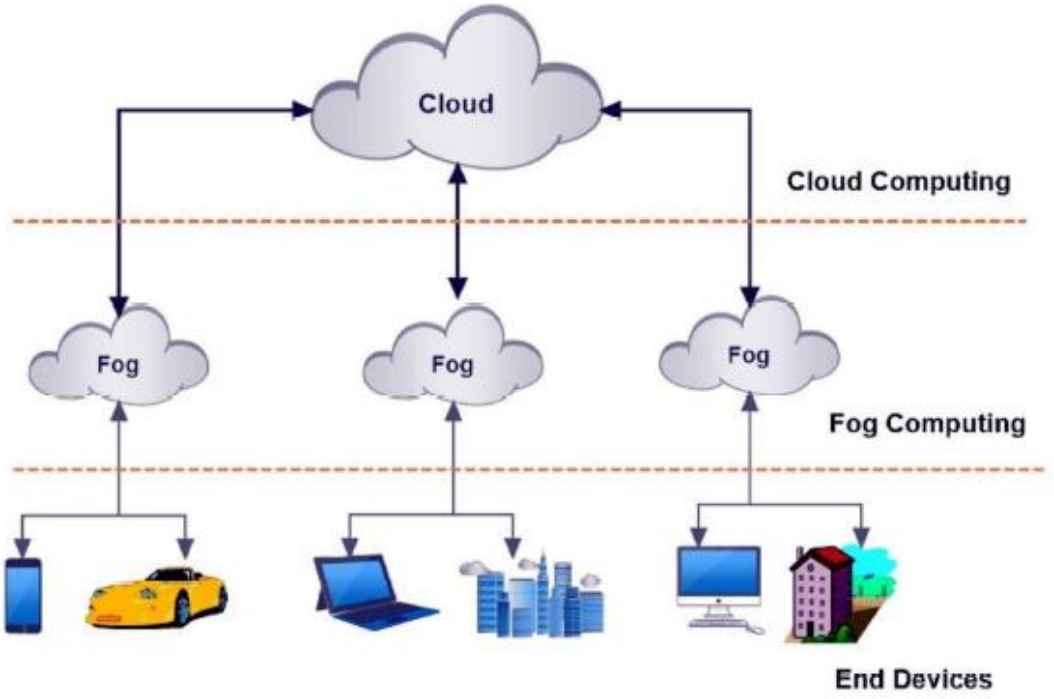

Figura 1: *Fog Computing* [1]

Esencialmente, la computación en la niebla es una extensión de la nube, pero que se encuentra más cerca de las cosas que trabajan directamente con los datos del IoT [1].

Como se muestra en la Figura 1, la computación en la niebla actúa como un intermediario entre la nube y los dispositivos finales, añadiendo servicios de red, almacenamiento y procesamiento de datos de forma más cercana a los dispositivos finales. Cualquier dispositivo con estas capacidades puede ser un nodo de esta red, como dispositivos de control industriales, conmutadores, encaminadores, sistemas embebidos, etc [1].

En general, el propósito de construir un entorno de estas características es recolectar y procesar datos de los dispositivos finales para analizar y encontrar patrones, o realizar análisis predictivos u optimizaciones para finalmente, tomar decisiones más inteligentes de una manera más adecuada. En estos entornos, lo datos se pueden clasificar en dos categorías [3]:

i) "***Little data***" o "***Big Stream***": Datos transitorios que se capturan de forma constante de los dispositivos de IoT.

ii) "***Big data***": datos y conocimientos persistentes almacenados y archivados en un almacenamiento centralizado en la nube.

Aunque la computación en la niebla usa recursos similares y comparte muchos mecanismos y atributos con la computación en la nube, trae muchos beneficios para los dispositivos IoT. Estos beneficios se pueden resumir en los siguientes:

i) **Mayor agilidad de negocio**: Con el uso de las herramientas correctas, las aplicaciones de computación en la nube pueden ser fácilmente desarrolladas y desplegadas. Además, estas aplicaciones pueden ser programadas para trabajar según las necesidades del cliente.

ii) **Baja latencia**: La computación en la niebla tiene la habilidad de soportar servicios en tiempo real.

iii) **Distribución geográfica a gran escala**: La computación en la niebla puede proveer de capacidad de almacenamiento y cómputo en aplicaciones distribuidas grandes.

iv) **Menor gasto operativo**: Se ahorra ancho de banda de la red ya que se procesan los datos seleccionados localmente en lugar de enviarlos a la nube para su análisis.

v) **Flexibilidad y heterogeneidad**: La computación en la nube permite la colaboración de distintos entornos físicos y e infraestructuras sobre múltiples servicios.

vi) **Escalabilidad**: La cercanía de la computación en la niebla a los dispositivos finales permite escalar el número de dispositivos conectados y servicios.

Respecto a la arquitectura de la computación en la nube, ésta consta de seis capas: capa física y virtualización, monitorización, preprocesamiento, almacenamiento temporal y capa de transporte, tal y como se muestran en la Figura 2.

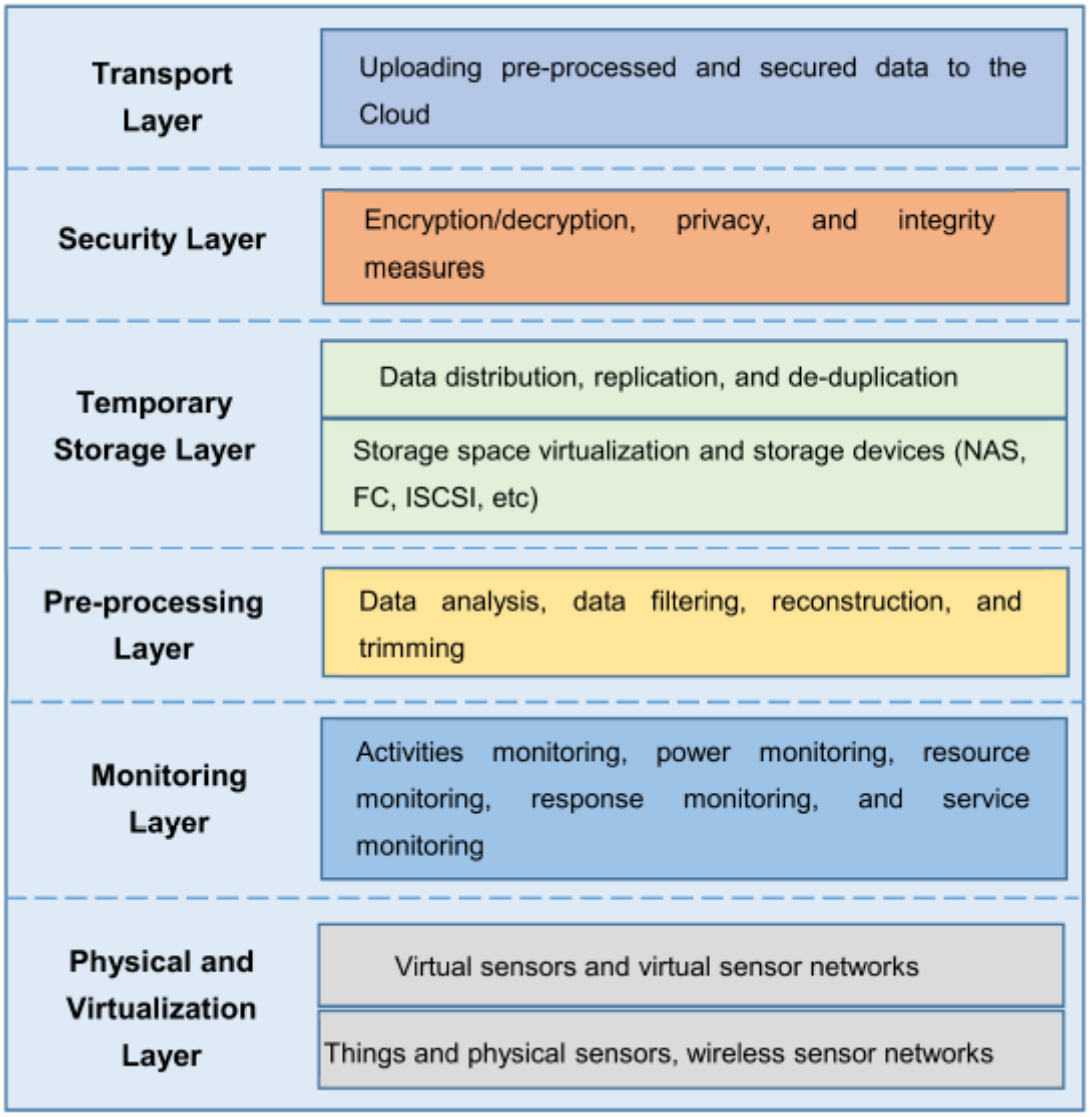

Figura 2: Arquitectura de la computación en la niebla [1]

La **capa física y de virtualización** implica diferentes tipos de nodos, como nodos físicos, nodos virtuales y redes de sensores virtuales. Estos nodos se gestionan y mantienen en función de sus tipos y demandas de servicio. Los diferentes tipos de sensores se distribuyen geográficamente para detectar los alrededores y envían los datos recogidos a las capas superiores a través de pasarelas para su posterior procesamiento y filtrado.

A continuación tenemos la **capa de monitorización**, en donde se supervisa la utilización de los recursos, la disponibilidad de los sensores y los nodos de niebla y los elementos de la red. En esta capa se supervisan todas las tareas realizadas por los nodos, supervisando qué nodo está realizando qué tarea, en qué momento y qué se le pedirá a continuación. Se supervisa el rendimiento y el estado de todas las aplicaciones y servicios desplegados en la infraestructura. Además, se supervisa el consumo de energía de los nodos de niebla.

La **capa de preprocesamiento** realiza tareas de gestión de datos. Los datos recogidos se analizan y se filtran y recortan en esta capa para extraer información importante. Los datos pre-procesados se almacenan temporalmente en la capa de almacenamiento temporal. Cuando los datos se transmiten a la nube, ya no es necesario almacenarlos localmente y pueden ser eliminados del medio de almacenamiento temporal.

En la **capa de seguridad**, entra en juego el cifrado/descifrado de los datos. Además, se pueden aplicar medidas de integridad a los datos para protegerlos de la manipulación.

Por último, en la **capa de transporte**, los datos pre-procesados se suben a la nube para que ésta pueda extraer y crear servicios más útiles. Para una utilización eficiente de la energía, sólo se sube a la nube una parte de los datos recogidos. En otras palabras, el dispositivo de pasarela que conecta el IoT con la nube procesa los datos antes de enviarlos a la nube. Este tipo de pasarela se denomina pasarela inteligente. Los datos recogidos por las redes de sensores y los dispositivos IoT se transfieren a la nube a través de pasarelas inteligentes. Los datos recibidos por la nube se almacenan y se utilizan para crear servicios para los usuarios. Teniendo en cuenta los recursos limitados de la computación en la niebla, un protocolo de comunicación debe ser eficiente, ligero y personalizable, por tanto, la elección del protocolo de comunicación depende del escenario de aplicación [1].

Cabe decir que, debido a las características de la computación en la niebla, tenemos distintos modelos de comunicación para sus dispositivos, siendo uno de los más

utilizados en la computación en la niebla la **Comunicación *Machine-To-Machine***. Este modelo de comunicación representa a distintos dispositivos que pueden conectarse de tal forma que intercambien información entre ellos de forma directa, sin ninguna de necesidad de que haya ningún dispositivo intermedio. Estos dispositivos son capaces de conectarse con otros no sólo a través de internet o redes IP, sino haciendo uso de otras tecnologías como Bluetooth. Este modelo es bastante utilizado en una gran cantidad de aplicaciones dentro de sistemas como hogares inteligentes, en donde estos dispositivos se comunican entre ellos a través de mandarse pequeños paquetes de datos con unos requisitos bastante sencillos [3].

Respecto a las posibles aplicaciones de la computación en la nuble, algunas de las más destacadas son las siguientes:

i) **Sanidad**: Hay algunas herramientas desarrolladas que permiten analizar en tiempo real parámetros clave del estado de salud de los pacientes, como la herramienta FAST [39], que es un sistema de análisis distribuido en un entorno de computación en la niebla cuya función es monitorizar las posibles caídas de los pacientes con ictus. Este sistema tiene una arquitectura definida con 3 niveles para una infraestructura sanitaria inteligente con el fin de proporcionar una arquitectura eficiente para aplicaciones sanitarias y de atención a la tercera edad. Estos 3 niveles de arquitectura son:

   1. Módulo *Front-End*: Ejecuta los algoritmos para recolectar los datos necesarios del paciente.
   2. Módulo *Back-end*: Este módulo se ejecuta en la nube y realiza las operaciones necesarias con los datos recolectados.
   3. Módulo de comunicación: Provee dos canales de comunicación entre el módulo *Front-End* y el *Back-End*.

ii) **Realidad aumentada**: Las aplicaciones de realidad aumentada son muy sensibles a una latencia alta, ya que incluso los retrasos más pequeños en el tiempo de respuesta pueden perjudicar la experiencia de usuario. Por ello, la computación en la nube tiene la potencia de convertirse en un importante actor en el ámbito de la realidad aumentada.

iii) **Almacenamiento en caché y preprocesamiento en la web**: Una posible mejorar es la interacción con la navegación Web, de tal forma que se pueda mejorar el rendimiento de los sitios web. El usuario realiza la petición HTTP a través de uno de los dispositivos finales de la computación en la niebla, y este dispositivo realiza una serie de optimizaciones que reduce el tiempo que el usuario espera para que se cargue la página web solicitada. Además de algunas optimizaciones genéricas, como el almacenamiento en caché de los componentes HTML o la reorganización de la composición de la página web, los dispositivos finales de la computación en la niebla también realizar optimizaciones que tienen en cuenta el comportamiento del usuario y las condiciones de la red [3].

2.2 Ordenadores monoplaca

Debido a la naturaleza de las arquitecturas de computación en la niebla, se debe de tener en cuenta el coste de los dispositivos que van a realizar funciones de nodo. Para ello, una opción económica es el uso de placas SBC.

Un computador de placa única o **Single Board Computer (SBC)** es un ordenador completo en la que una placa de circuito único contiene la memoria, un microprocesador, la entrada/salida, y todas las demás características necesarias para que se logre un correcto funcionamiento. A diferencia de un ordenador de sobremesa moderno, un SBC no necesita ni memoria ni almacenamiento adicional para arrancar, ni cables adicionales de alimentación para ningún componente. Tampoco necesita añadirle ninguna tarjeta RAM ni varios ventiladores de caja [8].

Estos ordenadores están diseñados de forma distinta a los ordenadores de sobremesa, ya que éstos son completamente autónomos. Normalmente cuentan con una amplia gama de microprocesadores y tienen una mayor densidad para los circuitos integrados utilizados.

El uso de este tipo de ordenadores de una sola placa tiene muchas ventajas, sus características están bien integradas debido a que casi todo es nativo de la máquina. A menudo se suelen proporcionar ranuras para la interconexión, y se dispone de configuraciones de ranuras y placas base. Además, puesto que son placas mucho más

compactas, son más eficientes en el consumo de energía que los ordenadores convencionales.

Sin embargo, los ordenadores de una sola placa también tienen sus limitaciones. Su formato estándar puede no ser adecuado para las necesidades de un cliente en particular. También pueden ser difíciles de utilizar para aplicaciones que requieren la eliminación de cables o el uso de conectores especiales de entrada/salida.

Su principal uso es en aplicaciones integradas, además de aplicaciones cuyo objetivo es el control de procesos, como los sistemas robóticos complejos y las aplicaciones de uso intensivo de procesadores.

Hay que realizar una distinción entre los SBC y una **MCU (Unidad de microcontrolador)**. Éstas últimas es una forma de sistema incorporado que incluye un ordenador entero en un único chip. Al contrario que los SBC, no son dispositivos potentes ni fácilmente reprogramables, sin embargo, para tareas sencillas que requieren poca capacidad de cómputo pueden ser ideales [8].

Dentro de todas las opciones de placas SBC que hay disponibles en el mercado (algunas de ella son la BeagleBone Black, la UDOO Quad, o la Odroid XU4 [8]) las **Raspberry Pi** son las que más éxito han tenido alrededor del mundo debido a su flexibilidad, simplicidad y multitud de accesorios compatibles. Tal y como se puede observar en la Figura 3, la Raspberry contiene todos los elementos necesarios para poder operar siendo una placa de espacio reducido, Originalmente, su uso estaba pensado para el ámbito educativo, pero gracias al movimiento “Do It Yourself”, dio un empuje al mundo de la

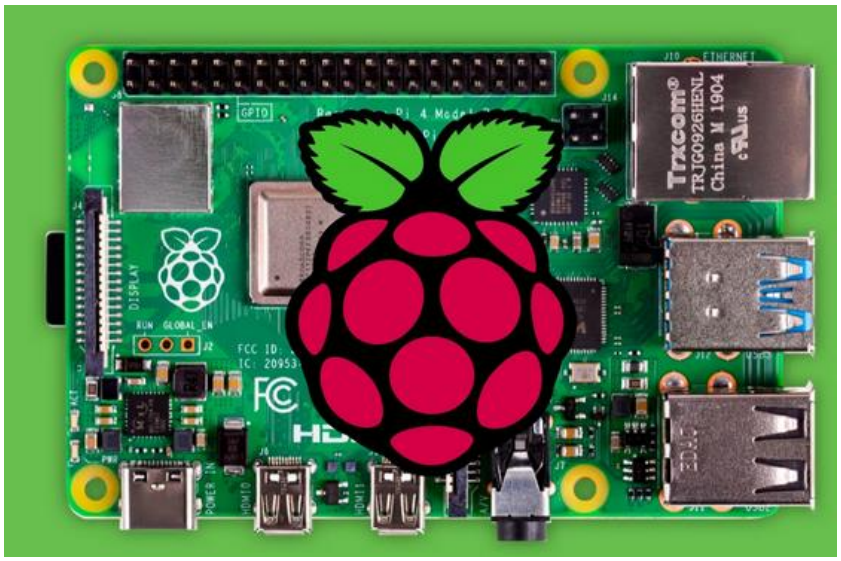

Figura 3: Raspberry Pi [13]

informática y generó una inmensa cantidad de proyectos con estos mini ordenadores siendo los principales protagonistas [9].

Algunos de estos proyectos que se pueden realizar con la Raspberry Pi son los siguientes:

i) **Mini PC de escritorio**: El uso más evidente que se le puede usar a una Raspberry PI es como un mini PC, ya que cuenta con todos los componentes en su placa para poder ejecutar un sistema operativo, así como varios puertos USB donde conectar los distintos periféricos, además de una conexión HDMI donde conectar un monitor.

ii) **Sistema de música en streaming casero:** Instalando el sistema de sonido en *streaming* "*Pi MusicBox*", este software convierte tu Raspberry Pi en un receptor de música en *streaming* compatible con Spotify otras aplicaciones de la misma índole.

iii) **Máquina de arcade retro:** Con sistemas operativos preparados para la emulación de una gran cantidad de videoconsolas retro, se puede convertir una Raspberry Pi en una consola de videojuegos versátil.

iv) **Domótica y sistemas de videovigilancia:** La existencia de plataformas de software libre para el control de dispositivos desde Raspberry Pi, como Domoticz, añaden compatibilidad con los principales estándares de conectividad entre dispositivos de domótica, haciendo mucho más sencilla la integración de nuevos dispositivos para el hogar conectado, como cámaras de videovigilancia que permitirían tomar fotos o grabar vídeo cuando se detecte movimiento en su rango visual [10].

2.3 Virtualización, contenedores y Docker

La **virtualización** es una tecnología que permite crear servicios de TI mediante recursos que están ligados de forma tradicional al hardware. Además, distribuye sus funcionalidades entre diversos usuarios o entornos, de tal forma que le permite utilizar todas las capacidades de una máquina física.

Aunque la tecnología de virtualización data de la década de los años 60, no fue hasta la década de los 90 en donde la virtualización tuvo un verdadero despegue, ya que, gracias a esta tecnología, las empresas podían compartir los servidores y ejecutar aplicaciones heredadas en varios tipos y versiones de sistemas operativos. Así, los servidores

empezaron a utilizarse de forma más eficiente, y, en consecuencia, se redujeron todos los costos relacionados con instalación, refrigeración, mantenimiento, etc. El uso generalizado de la virtualización redujo la dependencia de un solo proveedor y se transformó en la base de lo que hoy en día se conoce como *cloud computing* [4].

Un **hipervisor** es un software que se encarga de crear y ejecutar máquinas virtuales y que, además, aísla el sistema operativo y los recursos del hipervisor de las máquinas virtuales y permite crearlas y gestionarlas. El hipervisor usa los recursos (CPU, memoria y almacenamiento) como un conjunto de medios que pueden redistribuirse fácilmente entre las máquinas virtuales. Necesita algunos componentes a nivel del sistema operativo para poder ejecutar las máquinas virtuales, como el administrador de memoria, los controladores de dispositivos, la entrada/salida, etc. Al proporcionar estos recursos, puede gestionar la programación de ellos en función de los recursos físicos. El hardware físico sigue efectuando las operaciones, por lo que la CPU aún ejecuta las instrucciones de la CPU según lo solicitado por las máquinas virtuales. Con un hipervisor, muchos sistemas operativos diferentes pueden funcionar a la vez y compartir los mismos recursos de hardware virtualizados. Ésta es una de las claves de la virtualización, ya que, sin ella, sólo se podría ejecutar un sistema operativo en el hardware [5].

Sin embargo, hay ocasiones en las que no necesitaremos la virtualización de todo el sistema operativo, ya que estaríamos desaprovechando recursos del hardware. Para ello, tenemos presente los **contenedores de aplicaciones**, los cuales son entornos ligeros de tiempo de ejecución que proporcionan a las aplicaciones los archivos, las variables y las bibliotecas que necesitan para ejecutarse, maximizando de esta forma su portabilidad. A diferencia de las máquinas virtuales, los contenedores utilizan el sistema operativo de su host en lugar de proporcionar el suyo propio [6].

Además, cuando se habla de contenedores, debemos de tener en cuenta 2 conceptos muy importantes:

i) La **orquestación** se encarga de automatizar la programación, implementación, escalabilidad, equilibrio de carga, disponibilidad y redes de contenedores. Es decir, su función es automatizar y gestionar el ciclo de vida de los contenedores y servicios [27].

ii) La **gestión del ciclo de vida de las operaciones contenerizadas** se refiere a la parte de la tecnología que se centra en facilitar la implementación y la gestión operativa del ciclo de vida de las aplicaciones dentro de los contenedores. La finalidad de esta característica es ir más allá de lo que un contenedor hace, de tal forme que explote al máximo lo que un sistema de contenedores podría necesitar y hacer, con miras a las tecnologías basadas en la nube, los microservicios y otras existentes y nuevas, para facilitar la integración de los beneficios de las tecnologías de virtualización basadas en contenedores.

Con todo, algunas de las ventajas que ofrece este tipo de tecnología son las siguientes [6]:

i) Aumenta el rendimiento y eliminan el gasto de recursos de memoria y procesador al compartir un mismo sistema operativo base.

ii) Beneficia a los entornos de nube de múltiples maneras, ya que, en comparación con una máquina virtual, los contenedores son ambientes de componentes ligeros que permiten a las aplicaciones moverse en la nube sin necesidad de un gran trabajo por rehacerlas.

iii) Minimiza los recursos redundantes que cada instancia virtual necesita, permitiendo al mismo servidor alojar más contenedores que máquinas virtuales comparables, mejorando así la escalabilidad y el rendimiento sobre la nube.

iv) Mejora los entornos que exigen cómputo de gran escala y que comparten componentes clave.

v) Permite la arquitectura de microservicios, lo que los convierte en una nueva forma de ensamblar aplicaciones.

vi) Provee mayores velocidades de respuesta, ya que se pueden iniciar y detener mucho más rápido que las máquinas virtuales.

Sin embargo, este tipo de tecnología no está libre de algunas desventajas, como pueden ser que tiene menos versatilidad al depender de un solo sistema operativo comparado con la virtualización basada en hipervisores.

Una de las tecnologías más usadas en lo que respecta a la creación y uso de contenedores de Linux es **Docker** [7]. La tecnología Docker usa el kernel de Linux y las funciones de éste, como **cgroups** y **namespaces**, para segregar los procesos, de modo que puedan ejecutarse de manera independiente. Esta tecnología de contenedores ofrece un modelo de implementación basado en imágenes. Esto permite compartir una aplicación, o un conjunto de servicios, con todas sus dependencias en varios entornos. También automatiza la implementación de la aplicación (o conjuntos combinados de procesos que constituyen una aplicación) en este entorno de contenedores. Estas características son lo que hace a Docker fácil de usar y único, otorgan a los usuarios un acceso sin precedentes a las aplicaciones, la capacidad de implementar rápidamente un contenedor en cualquier momento, y, por último, control sobre las versiones y su distribución.

Al principio, la tecnología Docker se desarrolló a partir de la **tecnología LXC**, lo que la mayoría de las personas asocia con contenedores de Linux "tradicionales", aunque desde entonces se ha alejado de esa dependencia. LXC era útil como virtualización ligera, pero no ofrecía una buena experiencia al desarrollador ni al usuario. La tecnología Docker no solo aporta la capacidad de ejecutar contenedores; también facilita el proceso de creación y diseño de contenedores, de envío de imágenes y de creación de versiones de imágenes. En la Figura 4 puede apreciarse la diferencia entre ambos tipos de virtualización.

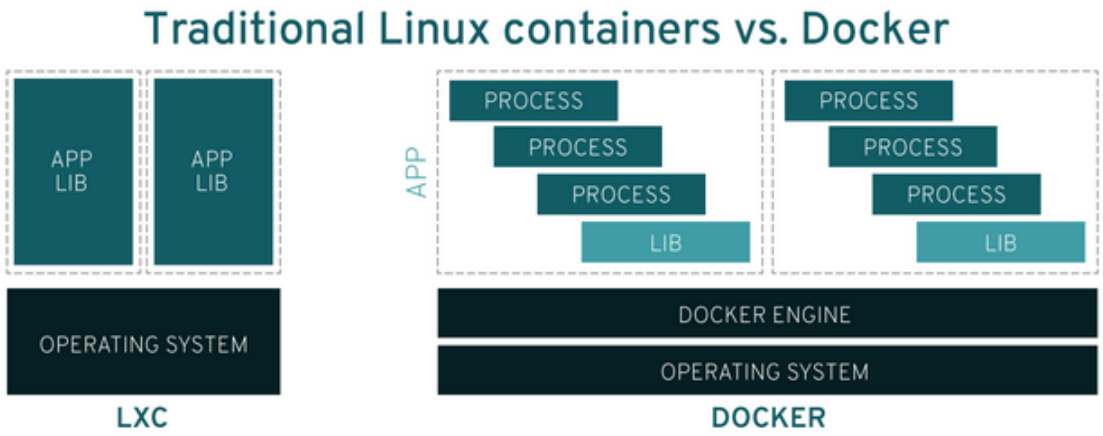

Figura 4: Diferencia entre LXC y Docker [7]

Los contenedores de Linux tradicionales usan un sistema `init` que puede gestionar varios procesos. Esto significa que las aplicaciones completas se pueden ejecutar como

una sola. La tecnología Docker pretende que las aplicaciones se dividan en sus procesos individuales y ofrece las herramientas para hacerlo. Este enfoque granular tiene sus ventajas.

El enfoque Docker para la creación de contenedores se centra en la capacidad de tomar una parte de una aplicación, para actualizarla o repararla, sin necesidad de tomar la aplicación completa. Además de este enfoque basado en los microservicios, puede compartir procesos entre varias aplicaciones de la misma forma que funciona la arquitectura orientada al servicio. Cada archivo de imagen de Docker se compone de una serie de capas. Estas capas se combinan en una sola imagen. Una capa se crea cuando la imagen cambia. Cada vez que un usuario especifica un comando, como ejecutar o copiar, se crea una nueva capa.

Docker reutiliza estas capas para construir nuevos contenedores, lo cual hace mucho más rápido el proceso de construcción. Los cambios intermedios se comparten entre imágenes, mejorando aún más la velocidad, el tamaño y la eficiencia. Además, El control de versiones es inherente a la creación de capas. Cada vez que se produce un cambio nuevo, básicamente, se tiene un registro de cambios incorporado [7].

2.4 Orquestación de contenedores

El auge de los contenedores ha cambiado la forma en la que los usuarios conciben el desarrollo, el despliegue y el mantenimiento de las aplicaciones de software. Haciendo uso de las capacidades de aislamiento nativo de los sistemas operativos modernos, pero sin consumir tantos recursos y con una mayor flexibilidad de despliegue. Puesto que los contenedores son tan ligeros y flexibles, éstos han dado lugar a nuevas arquitecturas de aplicaciones. Este nuevo enfoque consiste en empaquetar los diferentes servicios que constituyen una aplicación en contenedores separados, y luego desplegar esos contenedores a través de un clúster de máquinas físicas o virtuales. Sin embargo, en la actualidad, debido a la complejidad de las aplicaciones, no se despliegue un solo contenedor en producción. Lo habitual es necesitar varios, los cuales escalan de manera distinta y cada uno tiene sus complejidades. Por ello, llegamos a la necesidad de la **orquestación de contenedores**, es decir, disponer de herramientas que nos ayuden a automatizar el despliegue, la gestión, el escalado, la interconexión y la disponibilidad de

nuestras aplicaciones basadas en contenedores. Por ello, el orquestador de contenedores se ocupa de cuestiones como:

- Configuración automática.
- Despliegue e inicio automático de servicios basados en contenedores.
- Balanceado de carga.
- Auto-escalado y auto-reinicio de contenedores.
- Intercambio de datos y datos de networking.

Aunque Kubernetes es el orquestador de contenedores más conocido en el mundo, y es el orquestador usado durante este TFG, nombraremos algunos de los más usados:

i) **Nomad:** Es una herramienta de orquestación que permite desplegar diferentes tipos de aplicaciones, como basadas en contenedores o aplicaciones orientadas a microservicios [40].

ii) **Docker Swam:** Permite agrupar una serie de hosts de Docker en un clúster y gestionar los clústeres de forma centralizada, así como la orquestación de contenedores. Se basa en una arquitectura maestro-esclavo en donde hay un nodo maestro el cual es responsable de la gestión del clúster, y uno o varios nodos esclavos que se encargar de ejecutar las unidades de trabajo [41].

iii) **Docker Compose:** Herramienta que permite definir y agrupar múltiples aplicaciones utilizando contenedores Docker. Se usan archivos YAML que nos servirán para definir la configuración de la aplicación. Así, con un solo comando se crea e inician los servicios configurados en estos ficheros [50].

2.5 Kubernetes

Una vez descrito qué son los orquestadores de contenedores y algunos ejemplos de ellos, profundizaremos en el orquestador utilizado durante el transcurso de este TFG, Kubernetes.

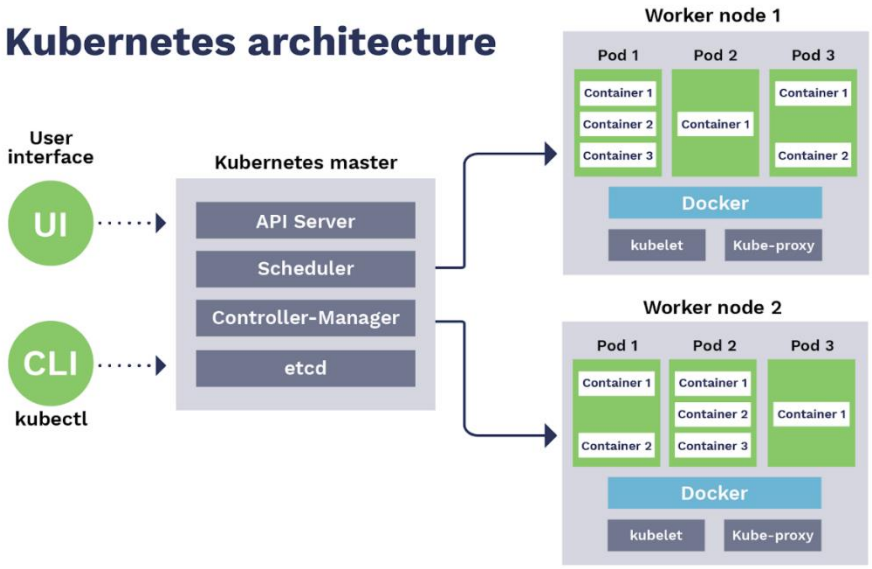

Figura 5: Arquitectura de Kubernetes [28]

Este orquestador, ilustrado en la Figura 5, nos ofrece un entorno de administración centrado en contenedores. Maneja la infraestructura de cómputo, redes y almacenamiento para que las cargas de trabajo de los usuarios no tengan que hacerlo [12].

En Kubernetes, la unidad más pequeña de trabajo es lo que se conoce como un pod. Un pod está formado por uno o más contenedores, en donde los contenedores que pertenecen al mismo pod comparten los recursos informáticos [15]. La Figura 6 muestra de forma gráfica el concepto de pod. Cada uno tiene una dirección IP interna dentro del clúster de Kubernetes.

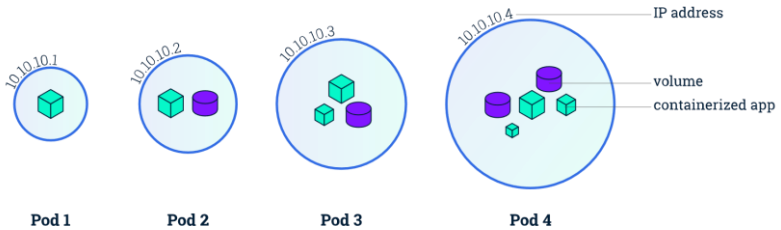

Figura 6: Pods [29]

Para representar las distintas entidades del sistema, utilizamos objetos, los cuales representan entidades persistentes dentro del sistema de Kubernetes [14]**.** Estos objetos pueden describir:

- Qué aplicaciones se ejecutan en contenedores, y en qué nodo se encuentran.
- Los recursos disponibles para las aplicaciones.
- Las políticas sobre cómo estas aplicaciones se comportan (reinicio, actualización, tolerancia a fallos).

El plano de control se encarga de mantener un registro de todos los objetos de Kubernetes **presentes** en el sistema y ejecuta continuos bucles de control para gestionar el estado de los mismos. Como podemos observar en la Figura 5, el plano de control consta de varios *daemons* que se ejecutan en el nodo master y otros tantos que se ejecutan en los nodos worker [42].

Estos Daemon son:

- **API Server**: Su función es validar y configurar los datos de los distintos objetos de Kubernetes, como los pods, replicationControllers, etc [43].
- **Controller Manager**: La función de este *Daemon* es controlar el estado del clúster mediante un bucle y que a través del API Server realiza los cambios para tener el estado deseado del clúster [44].
- **Scheduler**: Se encarga de asignar los pods a los distintos nodos del clúster. Será explicado con mayor profundidad en el punto 2.5.1 [45].
- **Etcd**: Es un almacén de datos persistente, consistente y distribuido, utilizado para almacenar toda la información del clúster de Kubernetes [48].
- **Kubelet**: Es el agente en cada nodo worker que se ejecuta en cada nodo del clúster e interacciona con el runtime de contenedores, como Docker. Se asegura que los contenedores estén corriendo en un pod [46].
- **Kube-proxy**: Se encarga de gestionar la comunicación de red dentro y fuera del clúster [47].

Además, Kubernetes cuenta con una herramienta llamada **namespaces**, que permite crear múltiples clústeres virtuales respaldados por el mismo clúster físico. Generalmente se usa en entornos con muchos usuarios distribuidos en múltiples equipos [49].

Una vez se indica a kubernetes el tipo y cantidad de objetos a desplegar, kubernetes se encargará de monitorizar el estado del clúster y mantener el mismo cuando se detecten fallos de funcionamiento en algunos de los objetos descritos. Cuando se crea un objeto, la realidad es que el usuario le está indicando al clúster cómo quiere que sea el estado deseado del clúster.

A la hora de definir un objeto, se debe de especificar la '**spec**' del objeto que describe su estado deseado, así como otra información básica (tipo de objeto, nombre, etc) que se detalla a continuación. Estas peticiones se realizan a la API de Kubernetes en formato JSON. Esta información es proporcionada a través de un archivo '**.yaml**', usando **kubectl** (interfaz de línea de comandos que ejecuta comandos sobre despliegues clusterizados para kubernetes). Un ejemplo de archivo yaml está incluido en el Anexo II.3, en el cual se realiza un despliegue de 2 contenedores que contienen una imagen del servidor web nginx.

Algunos de los campos obligatorios que se deben de incluir en los archivos yaml son el **apiVersion** (qué versión de la API vamos a usar), **kind** (qué tipo de objeto vamos a crear) y **metadata** (información que identifica unívocamente al objeto, en donde se incluyen datos como el **name** o el **namespace**) [14].

Algunos de los tipos de objetos más comunes son los llamados controladores, siendo los más usados los siguientes:

- **ReplicaSet:** La función de un `ReplicaSet` consiste en mantener un conjunto estable de réplicas de pods ejecutándose en todo momento. Así se puede garantizar la disponibilidad de un número específico de pods idénticos [30].
- **Deployment:** La función de un `Deployment` consiste en proporcionar actualizaciones declarativas para los pods y los `ReplicaSets`. Cuando se describe el estado deseado en un `Deployment`, el controlador del `Deployment` se encarga de cambiar el estado actual al estado deseado de forma controlada en el clúster [31].
- **StatefulSets:** Se encarga de gestionar el despliegue y el escalado de un conjunto de pods, que se basan en una especificación idéntica de contenedor [32].

Actualmente, Kubernetes es el estándar de facto para las herramientas de orquestación de contenedores. Es por ello que los principales proveedores de la nube ofrecen este producto como ***Kubernetes-as-a-Service*** [51]:

i) **Amazon EC2 Container Service (ECS)**: El servicio de AWS para orquestación de contenedores es un sistema de gestión muy escalable que permite a los desarrolladores ejecutar aplicaciones en contenedores sobre instancias EC2. Está formado por muchos componentes integrados que permiten la fácil planificación y despliegue de clústeres, tareas y servicios Docker [11].

ii) **Google Cloud Kubernetes:** Servicio proporcionado por Google para administrar clústeres de Kubernetes, ofreciendo una experiencia sin nodos, siendo Google el agente encargado de gestionar toda la estructura subyacente de todo el clúster [52].

iii) **Azure Kubernetes Service (AKS):** Servicio proporcionado por Microsoft Azure que permite administrar aplicaciones de Kubernetes de forma sencilla [53].

2.5.1 El planificador de Kubernetes

Para que Kubernetes sitúe los pods creados en los nodos del clúster, utiliza el llamado **planificador de Kubernetes**. Por cada pod que se crea, el planificador es responsable de encontrar el mejor nodo en el que encaje el pod para que funcione sobre él [16].

El planificador funciona como un pod en el nodo maestro, formando parte del plano de control de Kubernetes. El planificador, ilustrado en la Figura 7, funciona de la siguiente forma:

- El planificador mantiene una cola de pods llamada ***podQueue*** y se mantiene escuchando al ***APIServer***.
- Cuando se crea un pod, primero se escriben los metadatos del pod en **etcd** a través del ***APIServer***.
- Cada vez que se añade un pod nuevo, el pod se añade a la cola ***podQueue***.
- El proceso principal extrae de forma continua pods de la cola ***podQueue*** y los asigna a los nodos.

- El proceso de planificación consta de dos pasos:
    - Un filtro que pre-selecciona los nodos que no cumplen los requisitos, por ejemplo, ***disktype:ssd*** (Se debe de indicar en el archivo YAML, el host debe de tener un disco duro sólido).
    - De la lista de nodos que han pasado el primer filtro, a los nodos se les asigna una puntuación dependiendo de ciertas configuraciones y métricas, y se selecciona el nodo con mayor puntuación. Por ejemplo, una métrica prioriza las máquinas con mayor porcentaje de CPU libre (***LeastRequestedPriority***) o trata de minimizar el número de pods desplegados pertenecientes al mismo servicio en el mismo nodo (***SelectoSpreadPriority***).
- Después de que un nodo haya sido asignado de forma satisfactoria, se invoca la interfaz del ***APIServer*** e indica el nombre del nodo al pod asignado.
- Mientras todo este proceso ocurre, desde el ***kubelet*** del nodo asignado escucha el ***APIServer***, y si encuentra que un pod ha sido asignado a ese nodo, el ***dockerDaemon*** es invocado para ejecutar el pod.

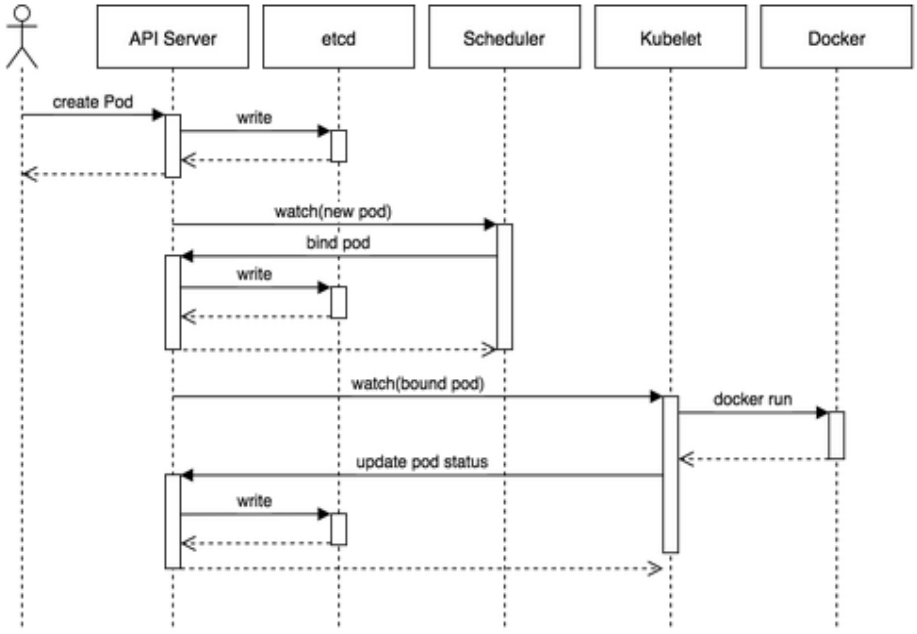

Figura 7: Fases del planificador [17]

El problema de este planificador que viene por defecto es que evalúa cada pod para cada uno de los nodos. Esta planificación por defecto es extremadamente lenta si tenemos un clúster con muchos nodos [17].

2.5.2 Desarrollo de nuevas funcionalidades en el planificador

Existen algunas opciones para mejorar la capacidad de planificación del planificador que viene por defecto. En primer lugar, podemos añadir *plugins* al planificador para añadirle las funcionalidades necesarias. Estos *plugins* son compilados dentro del propio planificador, mientras que el "núcleo" de dicho planificador se mantiene intacto [18]. Por otro lado, cabe la posibilidad de escribir un planificador totalmente hecho a medida según nuestras necesidades. Para ello, se debe de generar una imagen de Docker con el código del planificador y generar un pod con dicha imagen. Así, cuando queramos desplegar pods con este planificador personalizado, tan sólo tendremos que indicar el nombre del planificador en el archivo YAML del despliegue en el apartado 'pod.Spec.schedulerName'. Así, el planificador por defecto y el personalizado pueden convivir dentro del clúster y usar uno u otro según convenga [19].

2.5.3 Parámetros del planificador

A la hora de planificar un conjunto de pods, tenemos distintos tipos de restricciones dependiendo de si estas restricciones son a nivel de nodo, a nivel de *namespace*, o a nivel de pod.

A nivel de nodo, cuando se están ejecutando varias cargas de trabajo, existe la posibilidad de ejercer cierto control sobre qué cargas se pueden ejecutar en un grupo particular de nodos. Esto es lo que se conoce como ***taint* de nodo** (restricciones de nodo), en donde se marca un nodo de tal forma que el usuario del clúster evite o impida el uso del nodo para ciertos pods. Además, se cuenta con una característica complementaria llamadas las **tolerancias,** las cuales permiten designar qué pods se pueden usar en nodos que se encuentran "*tainted*".

Los efectos disponibles cuando se marca un nodo como "*tainted*" son los siguientes:

- **NoSchedule**: Los pods que no toleran esta *taint* no se ejecutan en el nodo.
- **PreferNoSchedule**: El planificador evita la programación de pods que no toleran esta *taint*.
- **NoExecute**: El pod se desaloja del nodo si ya está en ejecución en este y no está programado en el nodo si aún no está en ejecución en él.

Cabe la posibilidad de que haya algunos pods que toleren todas las *taints*, por tanto, no pueden ser desalojados [20].

Para poder poner estas *taints* a los nodos, debe hacer uso del siguiente comando:

```
kubectl taint nodes k8s-worker-1 kubernetes.io/hostname=k8s-worker-1:NoExecute
```

De forma análoga, para quitar esta *taint* se emplea el mismo comando, pero añadiéndole un guion al final:

```
kubectl taint nodes k8s-worker-1 kubernetes.io/hostname=k8s-worker-1:NoExecute-
```

Junto a esta posibilidad de marcar nodos, podemos cambiar las políticas de planificación de planificador, de tal forma que a la hora de desplegar un conjunto de pods, la política de planificación se combine con los nodos "*tainted*" de tal forma que éstos pods se puedan desplegar de la forma más optima posible [21].

A nivel de *namespace*, para obligar a que el clúster siga estás buenas prácticas de asignar correctamente la CPU y la memoria correspondiente a los pods, es posible establecer los *limits* y *requests* por defecto de un *namespace* a través de un objeto llamado ***LimitRange***, en donde se indican los máximos y los mínimos permitidos. Este objeto puede combinarse con los ***ResourceQuota***, los cuales permiten indicar cuales van a ser por defecto los *requests* y los *limits* de un pod, de tal forma que se puede restringir cuantos recursos pueden utilizar los pods que se encuentran en el *namespace* del *resourceQuota* [23].

A nivel de nodo, es posible que haya pods que consuman más recursos que otros pods para realizar su ejecución, para ello, se puede indicar cuanta CPU y cuanta memoria se le asigna para ejecutarse.

En primer lugar, tenemos la sección ***requests*** dentro del apartado ***resources***, en donde se define cuanta CPU y cuanta memoria va a necesitar como mínimo para ejecutarse. En segundo lugar, tenemos la sección ***limits***, también dentro del apartado *resources*, en donde se define el máximo de CPU y de memoria que va a poder usar el pod a lo largo de su ejecución. En la Figura 8 tenemos un ejemplo de cómo se debe de declarar dentro del fichero de configuración yaml.

```yaml
containers:
 - name: container1
   image: busybox
   resources:
     requests:
       memory: "32Mi"
       cpu: "200m"
     limits:
       memory: "64Mi"
       cpu: "250m"
```

Figura 8: Resources yaml [23]

Cuando un nodo puede tener pods que suman más del 100% de su capacidad, esto significa que puede llegar el momento en el que no tenga recursos disponibles para todos los pods. Llegado este caso, Kubernetes necesita conocer sus prioridades, en donde categoriza a los pods en tres clases llamadas **Quality of Service (QoS)**:

- ***BestEffort***: Son los pods que no tienen ni *requests* ni *limits*, por lo que serían los primeros en ser eliminados si fuese necesario.
- ***Guaranteed***: Son los pods que sus *requests* son iguales a sus *limits*. Así, se garantiza su supervivencia.
- ***Burstable***: Se encuentran entre medias de los *BestEffort* y los *Guaranteed*, sus *limits* no se corresponden con sus *requests* y puede que no estén ambos valores establecidos.

Es importante definir bien estos valores, ya que se pueden eliminar pods de forma indeseada en un momento crítico [22].

Capítulo 3

Metodología y Desarrollo

Durante el desarrollo de este capítulo, se describirá el clúster desplegado y se detallarán las distintas características del planificador de Kubernetes. Además, se definirán las distintas pruebas realizadas sobre el clúster para evaluar las capacidades del planificador.

3.1 Arquitectura desplegada

En este trabajo se plantea simular una arquitectura de computación en la niebla, en donde tenemos 3 Raspberry Pi a modo de nodos fog. Estos dispositivos son el ejemplo perfecto de placas SBC económicas y con una potencia que se ajusta a nuestras necesidades de cómputo. Puesto que el nodo maestro de nuestro clúster necesita ciertos recursos mínimos en el hardware para poder ejecutar todo el software necesario del plano de control de Kubernetes, usaremos una Raspberry Pi 4 modelo B de 4Gb de RAM, cuyas especificaciones son las ilustradas en la Tabla 1, mientras que para los nodos esclavos usaremos dos Raspberry Pi 3 modelo B, cuyas especificaciones se muestran en la Tabla 2 de forma análoga a la Raspberry Pi 4.

Aunque la gran mayoría de las arquitecturas de computación en la niebla envian los datos que hayan recopilado en los nodos a un servidor central en la nube, en lo que respecta a este TFG sólo nos centraremos en la gestión de nodos *fog*.

Arquitectura	**ARM v8**
Procesador	Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
Memoria RAM	1 GB RAM
Almacenamiento	MicroSD
Puertos USB	4
Alimentación	5 V
Red	BCM43438 wireless LAN 100 Base Ethernet
Tamaño	85 x 56 x 17 mm
Peso	45 g

Tabla 1: Especificaciones Raspberry Pi 3 Modelo B [37]

Arquitectura	**ARM v8**
Procesador	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
Memoria RAM	4GB RAM
Almacenamiento	MicroSD
Puertos USB	4
Alimentación	5 V
Red	2.4 GHz and 5.0 GHz IEEE 802.11ac wireless Gigabit Ethernet
Tamaño	85 x 56 x 17 mm
Peso	45 g

Tabla 2: Especificaciones Raspberry Pi 4 Modelo B [38]

Respecto al despliegue de aplicaciones, se ha optado por realizar los despliegues sobre una tecnología de virtualización ligera basada en contenedores como es Docker. Una vez elegido el hardware sobre el que vamos a trabajar y qué tecnología de virtualización vamos a usar, necesitamos un orquestador que nos permita realizar una gestión lo más **óptima** posible de nuestras aplicaciones. Para ello, se decidió por utilizar Kubernetes, ya que es considerado uno de los estándares de facto en cuanto a orquestación de contenedores. En la Tabla 3 se especifican las versiones del conjunto de software utilizado en el transcurso del TFG.

Sistema operativo	Raspbian
OS-Virtualization	Docker 20.10.6
Orquestador	Kubernetes 1.21.0
Benchmark	Sysbench

Tabla 3: Software instalado

En la Figura 9 se puede observar un esquema de la arquitectura desplegada, que consta de 3 Raspberry Pi conectadas a una red local, en donde cada placa SBC tiene instalado Raspbian 10 (Buster).

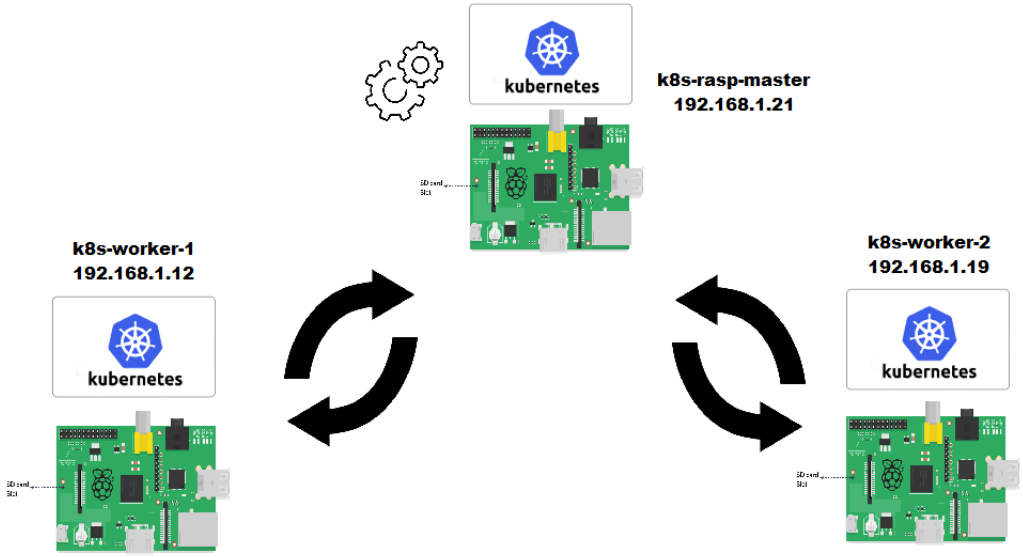

Figura 9: Arquitectura desplegada

En primer lugar, se ha desactivado el modo gráfico de los sistemas operativos, para ahorrar recursos y puesto que se va a acceder a la consola de los dispositivos mediante SSH, no se le va a dar uso a dicha interfaz. Una vez tengamos las placas SBC funcionando, y se haya instalado todo el software necesario para tener Docker y Kubernetes, se ha de

considerar qué imágenes vamos a usar. Puesto que en las pruebas se pretende analizar tiempos de ejecución, es necesaria una aplicación que en nuestro caso sea intensiva en el uso de CPU. Para ello, se ha utilizado un *benchmark* llamado Sysbench [33], que analizaremos con profundidad en el apartado 3.3.

3.2 Proceso de implementación del clúster

En esta sección abordaremos el proceso de implementación y configuración del clúster con Kubernetes. Previamente a la instalación de Kubernetes, debemos de preparar las máquinas para una correcta ejecución de la herramienta. En primer lugar, instalaremos el sistema operativo Raspbian en su última versión (Buster), y, habiendo conectado la memoria SD a un ordenador, accederemos al sistema de ficheros y habilitaremos SSH mediante archivo de texto. Además, cambiamos el ***hostname*** de la máquina por el que le corresponda (k8s-master-rasp4 en el caso de la máquina maestra y k8s-worker1/2 en caso de las máquinas esclavas). Puesto que vamos a montar el clúster en una red local privada, de cara a facilitar la gestión de los dispositivos, asignaremos IPs estáticas a cada dispositivo.

Una vez tengamos las tres placas preparadas, ejecutaremos el *script* indicado en el Anexo I.1, el cual se encarga de realizar las ultimas configuraciones para que el nodo maestro pueda operar de forma correcta y los nodos *worker* se puedan unir al clúster. Tras la ejecución de los scripts de instalación, para comprobar que no ha habido ningún problema, ejecutamos el comando:

```
$ kubeadm version
```

Si todo el proceso ha sido correcto, obtendremos una salida como la que podemos ver en la Figura 10:

```
pi@k8s-rasp4-master:~ $ kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"21", GitVersion:"v1.21.0", GitC
ommit:"cb303e613a121a29364f75cc67d3d580833a7479", GitTreeState:"clean", BuildDat
e:"2021-04-08T16:30:03Z", GoVersion:"go1.16.1", Compiler:"gc", Platform:"linux/a
rm"}
```

Figura 10: Salida del comando kubeadm

Destacar que en este proceso de instalación del sistema nos encontramos con varios problemas debido a las compatibilidades y dependencias de las librerías empleadas por Kubernetes, por lo que fue necesario actualizar el *firmware* de las Raspberry PI para

poder solucionar estos problemas iniciales. La continua evolución de Kubernetes y la gran actividad en el desarrollo de su código hacen que la documentación existente no está plenamente actualizada lo que genera este tipo de problemas. En este caso fue de gran ayuda la consulta de los fotos y blogs de los equipos de trabajo. Tras comprobar que toda la instalación ha sido correcta, debemos de lanzar kubeadm en el nodo maestro. Este comando se encargará de iniciar el resto de servicios de Kubernetes:

```
$ sudo kubeadm init
```

Finalizado el proceso, kubeadm nos dará un comando que permitirá unir a los nodos workers al clúster:

```
$ sudo kubeadm join 192.168.1.21:6443 –token
nz3vkt.ytg3t710m0frrddr \
  --discovery-token-ca-cert-hash
sha256:37ca90ab73df7a8888ee1d5c8092bf767074761ac0cb89c210a8bd
7f99d3c0fa
```

Este comando se debe de ejecutar desde cada nodo que queremos unir al clúster. Una vez se hayan unido todos los nodos, para verificar que los nodos están funcionando correctamente, ejecutamos el comando:

```
Kubectl get node
```

Si todo ha ido bien, se debe obtener la salida que se muestra en la Figura 11, en la que se puede ver como los tres nodos del clúster están activos y el nodo k8s-rasp4-master asume el rol de nodo maestro de cara a la gestión del sistema:

```
pi@k8s-rasp4-master:~ $ kubectl get node

NAME              STATUS   ROLES                  AGE   VERSION
k8s-rasp4-master   Ready    control-plane,master   35d   v1.21.0
k8s-worker-1       Ready    <none>                 34d   v1.21.0
k8s-worker-2       Ready    <none>                 34d   v1.21.0
```

Figura 11: Nodos del clúster

3.3 Aplicación usada

La aplicación que será usada a lo largo de todas las pruebas será Sysbench [33], cuya función será introducir carga en el *testbed*. Este *benchmark* es una herramienta de

ejecución multihilo, y está enfocado a pruebas de procesador central, test de transmisión de memoria, benchmark de lectura/escritura de disco, etc. En nuestro caso, sólo haremos pruebas empleando tiempo de CPU. La idea de este benchmark es lanzar un hilo único de cálculo de números primos, que se debe de indicar por parámetro. En la Figura 12, podemos observar cómo es la ejecución de este *benchmark.*

```
pi@k8s-rasp4-master:~ $ sysbench --test=cpu --cpu-max-prime=5500 run
sysbench 0.4.12:  multi-threaded system evaluation benchmark

Running the test with following options:
Number of threads: 1

Doing CPU performance benchmark

Threads started!
Done.

Maximum prime number checked in CPU test: 5500


Test execution summary:
    total time:                          39.7434s
    total number of events:              10000
    total time taken by event execution: 39.7358
    per-request statistics:
         min:                                  3.86ms
         avg:                                  3.97ms
         max:                                 31.59ms
         approx.  95 percentile:               4.03ms

Threads fairness:
    events (avg/stddev):           10000.0000/0.00
    execution time (avg/stddev):   39.7358/0.00
```

Figura 12: Salida ejecución sysbench

Como Sysbench está adaptado por defecto a la arquitectura x86-64, debemos de generar una imagen para ARM. Para ello, se creó un repositorio en Dockerhub en el cual se almacena la imagen de este benchmark [34]. En el Anexo III.1 se incluye el Dockerfile que se encarga de montar esta imagen. Para crear la imagen Docker, necesitaremos ejecutar el siguiente comando:

```
docker build -t angel96eur/sysbench .
```

Para comprobar que la imagen ha sido creada correctamente, la salida del comando `docker images` tal como se muestra en la Figura 13:

```
pi@k8s-rasp4-master:~ $ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
angel96eur/sysbench         latest              ddd5b7f6dc0d        3 months ago        155MB
```

Figura 13: Docker images sysbench

3.4 Escenarios

Con el objetivo de medir el comportamiento del planificador de Kubernetes, se han definido varios escenarios de pruebas, en los que lanzaremos varios despliegues de pods con distintas características y realizaremos un seguimiento para más tarde realizar el análisis correspondiente de los resultados obtenidos. Todas las pruebas están realizadas sin ninguna carga adicional, es decir, sin ningún proceso en segundo plano que pudiera interferir:

- En el **primer escenario** se trata de realizar comparativas de tiempos de ejecución entre despliegues con distinto número de pods, en donde cada despliegue se caracteriza por tener una de las 3 posibles clases de Calidad de Servicio (*BestEffort*, *Burstable*, *Guaranteed*).
- En el **segundo escenario** se trata de comprobar cómo distribuye el planificador los pods cuando se cambian las políticas de planificación en los nodos, realizando distintas combinaciones entre algunas opciones como *NoExecute* (No ejecutar pods en ese nodo) o *PreferNoSchedule* (mientras haya otras opciones para planificar, se busca un nodo distinto).
- En el **tercer escenario**, el sistema está sobrecargado, teniendo distintos despliegues simultáneos en los que cada pod tiene unas solicitudes de *limits* y *requests* distintos. Así, tratamos de comprobar si teniendo un sistema sobrecargado de pods cuyas clases son *BestEffort* y *Burstable*, además de pods *Guaranteed*, el planificador es capaz de expropiar recursos a pods cuya calidad de servicio es *BestEffort* para no perjudicar a los pods que sean desplegados con una calidad de servicio que sea *Guaranteed*.

Cabe decir que, en todos estos escenarios, hemos considerado que cada *pod* desplegado está formado por un único contenedor.

3.5 Métricas de evaluación

En lo que respecta a las métricas usadas, principalmente se han usado los tiempos de ejecución obtenidos tras terminar la ejecución de los *pods* de Sysbench, los cuales, en cada escenario planteado, se realizan 3 ejecuciones iguales y los datos usados para la elaboración de gráficas será la media de estas 3 ejecuciones. Además, en los escenarios

en los que estemos probando distintas políticas de planificación, el dato que usaremos será el nodo en el que está el *pod* ejecutándose, tal y como se puede observar en la Figura 14.

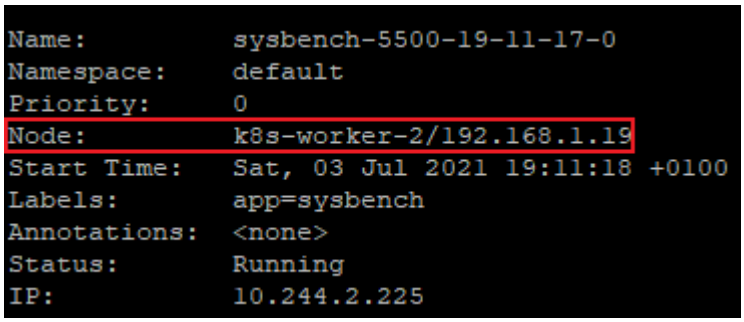

Figura 14: Asignación de pod en un nodo

3.6 Carga aplicada

Una vez planteados los distintos escenarios, vamos a utilizar dos scripts de *shell bash* para automatizar el lanzamiento de los despliegues de pods y la recolección de datos. En primer lugar, para el lanzamiento de pods utilizaremos el script cuyo pseudocódigo se muestra en la Figura 15 (El código completo se muestra en el Anexo I.2), el cual básicamente admite varios parámetros (archivo *yaml* del despliegue, número de pods que tiene el despliegue, número primo que calcula cada pod y valor de sleep por si se desea que haya una cierta cantidad de tiempo entre
el lanzamiento de cada pod). A continuación, dentro de un bucle `for` con tantas iteraciones como el número de pods que queremos lanzar, se realiza el despliegue de los pods.

```
script 1
  input: archivo yaml, nº de pods, nº primo, sleep
  output: pods desplegados
  {
     Se realiza una copia del archivo yaml
     for (tantas iteraciones como nº de pods) do
         Se obtiene el tiempo actual y se guarda en una variable
         Se modifica el fichero copia del yaml original para que cada pod tenga un nombre único
         Se ejecuta el despliegue del yaml
         se revierten los cambios en el archivo yaml copia
         Sleep del tiempo indicado en el parámetro
     done
     se elimina el archivo copia
  }
```

Figura 15: Pseudocódigo script 1

Cabe decir que en cada iteración se modifica el nombre de cada pod, en donde se añade la hora exacta a la que es lanzado, evitando así problemas de duplicidad en nombres. En la Figura 16 se muestra un ejemplo de despliegue. En él, se lanzan 5 pods cada 20

segundos. Aunque estos 20 segundos no son siempre exactos, ya que el propio script puede hacer que el lanzamiento entre dos pods tarde algún segundo más debido a las operaciones contenidas en él.

```
pi@k8s-rasp4-master:~/pruebasAutomatizacion $ ./script1.sh sysbench-pod.yaml 5 5500 20
despliegue: sysbench-pod.yaml
cantidad de pods: 5
numero primo: 5500
valor de sleep: 20
pod/sysbench-5500-00-42-18-20 created
pod/sysbench-5500-00-42-40-20 created
pod/sysbench-5500-00-43-01-20 created
pod/sysbench-5500-00-43-23-20 created
pod/sysbench-5500-00-43-44-20 created
```

Figura 16: Ejecución script de despliegue

Una vez se haya completado la ejecución del despliegue, haremos uso del script ilustrado en la Figura 17 (Código completo disponible en el Anexo I.3).

```
script 2
  input: -
  output: archivo csv con los resultados de la ejecución
  {
    Se guarda la salida de "kubectl get pods" en un archivo de texto
    Se lee el archivo y se guarda en otro archivo de texto los pods que estén "Completed"
    Se elimina el primer archivo de texto
    Se obtiene la cantidad de pods "Completed" para saber el tamaño del bucle
    for (cantidad de pods en estado "Completed") do
           Se coge la primera línea del archivo
           Se obtiene el log del pod para saber su tiempo de ejecución
           Se obtiene el nodo en el que está ejecutandose el pod
           Se obtiene el nombre del nodo
           Se guarda estas 3 variables en una misma línea
           Se elimina la primera línea del archivo
    done
    while (el archivo donde tenemos los datos siga teniendo líneas) do
          Se cogen las 3 primeras líneas y se ponen en el formato necesario para que sea un archivo csv
          Se eliminan las 3 primeras líneas del archivo
          Se actualiza el tamaño del bucle
    done
   }
```

Figura 17: Pseudocódigo script 2

Básicamente se encarga de obtener los pods cuyo estado sea “Completed”, comprobar el tiempo que ha tardado en ejecutarse y añadirlo a un archivo csv. En este archivo csv tendremos en cada línea el pod, el tiempo que ha tardado en ejecutarse y el nodo en el que fue planificado. Como puede observarse en la Figura 18, el script de recogida de datos utiliza el comando “kubectl get pods” para comprobar qué pods se encuentran en el estado “Completed”. Una vez los haya identificado, filtra los resultados convenientes y borra este pod de la lista.

```
pi@k8s-rasp4-master:~/pruebasAutomatizacion $ ./script2.sh
NAME                                READY   STATUS      RESTARTS   AGE
nginx-random-8584f6b7b-5tggj        0/1     Pending     0          94m
nginx-random-8584f6b7b-7xdpj        0/1     Pending     0          94m
sysbench-5500-00-42-18-20           0/1     Completed   0          2m35s
sysbench-5500-00-42-40-20           0/1     Completed   0          2m14s
sysbench-5500-00-43-01-20           0/1     Completed   0          112s
sysbench-5500-00-43-23-20           1/1     Running     0          91s
sysbench-5500-00-43-44-20           1/1     Running     0          70s
pod "sysbench-5500-00-42-18-20" deleted
pod "sysbench-5500-00-42-40-20" deleted
pod "sysbench-5500-00-43-01-20" deleted
pi@k8s-rasp4-master:~/pruebasAutomatizacion $ cat resultados.csv
Nombre;Tiempo;Nodo
sysbench-5500-00-42-18-20;87.6808;k8s-worker-1/192.168.1.12
sysbench-5500-00-42-40-20;88.2452;k8s-worker-2/192.168.1.19
sysbench-5500-00-43-01-20;88.3698;k8s-worker-2/192.168.1.19
```

Figura 18: Resultados obtenidos por el script 2

Este proceso de despliegue y recogida de datos ha sido igual para todos los escenarios, con la diferencia de que, en algunos casos, ha sido necesario ejecutar durante el tiempo de ejecución del despliegue el comando “kubectl get pods” para poder identificar el estado de cada pod.

3.7 Implementación de un planificador alternativo

Aunque el eje central de éste TFG es el estudio del comportamiento del planificador por defecto, como se ha nombrado anteriormente existen otras opciones para planificar pods. En este caso en concreto, se va a desplegar un planificador denominado “Random-scheduler” [35] el cual elige un nodo cualquiera de nuestro clúster de forma aleatoria. Este planificador no va a ser usado para hacer pruebas, sólo para ver cómo puede usarse un planificador distinto al que tenemos por defecto. Una vez tengamos la imagen de Docker, tal y como se muestra en la Figura 19, debemos de realizar un *deployment* [36] con esta imagen para tener un *pod*.

```
pi@k8s-rasp4-master:~/random-scheduler $ docker images
REPOSITORY                          TAG       IMAGE ID       CREATED         SIZE
angel96eur/randomscheduler          latest    e828fa8d7f91   5 weeks ago     34.1MB
<none>                              <none>    2e72e33544f0   5 weeks ago     808MB
```

Figura 19: Imagen Docker del planificador alternativo

Cuando se haga un despliegue que se quiera por este planificador alternativo, se debe de incluir la directiva indicada en la Figura 20 en el archivo YAML correspondiente:

```
spec:
  schedulerName: random-scheduler
```

Figura 20: Llamada al planificador alternativo

Así, cuando se realice el despliegue de pods en los nodos, serán planificados a través del pod que contiene la imagen del "random scheduler".

El código empleado para el despliegue de este planificador está disponible en *GitHub* [36]. Cuando se realiza el despliegue del *pod* que contiene este planificador alternativo, para comprobar que éste funciona correctamente, se debe de comprobar el log del *pod*. Para ello, se debe de realizar el proceso tal y como se ilustra en la Figura 21:

```
pi@k8s-rasp4-master:~/random-scheduler/deployment $ kubectl apply -f deployment.yaml
deployment.apps/random-scheduler created
pi@k8s-rasp4-master:~/random-scheduler/deployment $ kubectl get pods
NAME                               READY   STATUS              RESTARTS   AGE
random-scheduler-68dd8bb7-ssxxr    0/1     ContainerCreating   0          4s
pi@k8s-rasp4-master:~/random-scheduler/deployment $ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
random-scheduler-68dd8bb7-ssxxr    1/1     Running   0          2m47s
pi@k8s-rasp4-master:~/random-scheduler/deployment $ kubectl logs random-scheduler-68dd8bb7-ssxxr
I'm a scheduler!
```

Figura 21: Despliegue del planificador alternativo

Una vez tengamos el *pod* que contiene el planificador alternativo en estado *running*, podemos realizar despliegues de *pods* llamando a este planificador. Debido al funcionamiento interno de este planificador, en donde en la primera fase de filtrado se realiza un random para elegir que nodos pasan a la fase de clasificación, y en esta segunda fase, se realiza otro random en el que se le asigna un número aleatorio a cada nodo y se elige que tenga el número mayor de los asignados. Por ello, cabe la posibilidad de que pueda haber *pods* en estado *pending* hasta que el proceso que se encarga de realizar todo el proceso de planificación realice el despliegue correctamente. Una vez haya ocurrido esto, se puede comprobar en el log del *pod* del planificador cómo ha sido la planificación de estos pods. En la Figura 22 se ilustra el contenido de este log.

```
pi@k8s-rasp4-master:~/random-scheduler/deployment $ kubectl logs random-scheduler-68dd8bb7-ssxxr
I'm a scheduler!
2021/03/30 14:43:53 New Node Added to Store: k8s-master-rasp4
2021/03/30 14:43:53 New Node Added to Store: k8s-worker-1
2021/03/30 14:43:53 New Node Added to Store: k8s-worker-2

found a pod to schedule: default / sysbench-5500-22-13-04-0
2021/03/30 14:47:16 nodes that fit:
2021/03/30 14:47:16 k8s-worker-2
2021/03/30 14:47:16 k8s-master-rasp4
2021/03/30 14:47:16 calculated priorities: map[k8s-worker-2:80 k8s-master-rasp4:96]
Placed pod [default/sysbench-5500-22-13-04-0] on k8s-master-rasp4

found a pod to schedule: default / sysbench-5500-22-13-05-0
2021/03/30 14:47:26 nodes that fit:
2021/03/30 14:47:26 k8s-worker-1
2021/03/30 14:47:26 k8s-worker-2
2021/03/30 14:47:26 calculated priorities: map[k8s-worker-1:78 k8s-worker-2:95]
Placed pod [default/sysbench-5500-22-13-05-0] on k8s-worker-2

2021/03/30 15:10:28 nodes that fit:
2021/03/30 15:10:28 k8s-worker-1
found a pod to schedule: default / sysbench-5500-22-13-08-0
2021/03/30 15:10:28 calculated priorities: map[k8s-worker-1:84]
Placed pod [default/sysbench-5500-22-13-08-0] on k8s-worker-1

found a pod to schedule: default / sysbench-5500-22-13-09-0
2021/03/30 15:10:28 nodes that fit:
2021/03/30 15:10:28 k8s-master-rasp4
2021/03/30 15:10:28 k8s-worker-1
2021/03/30 15:10:28 calculated priorities: map[k8s-worker-1:2 k8s-master-rasp4:81]
Placed pod [default/sysbench-5500-22-13-09-0] on k8s-master-rasp4
```

Figura 22: log del planificador alternativo

Capítulo 4

Experimentos y Resultados

En este capítulo, se presentarán y evaluarán los resultados experimentales obtenidos en las pruebas planteadas.

4.1 Resultados del primer escenario planteado

Como se ha comentado en el capítulo anterior, en este primer escenario se va a realizar un conjunto de pruebas en la que se van a realizar una comparativa sobre los tiempos de ejecución de *pods* con distintas configuraciones de calidad de servicio para distintas cargas. En este primer conjunto de gráficas, se muestra el tiempo medio de la ejecución de distintos *pods*.

Como primera prueba, se lanzaron 10 despliegues de 1 a 10 contenedores sin ningún tipo de *limits* ni *requests* (configuración *BestEffort*) y otros 10 despliegues de 1 a 10 contenedores, pero con una configuración *Guaranteed* (mismo valor para *limits* y *requests*) en donde cada *pod* se asegura tener 0,75 unidades de CPU. Durante todo el experimento, el valor de sleep del *script* usado ha sido 0, para comprobar el comportomaniento del sistema en el peor caso. Como podemos observar en la Figura 23, se muestra el tiempo de ejecución medio de cada uno de los contenedores en cada uno de los despliegues realizados. Al asignar menos de una CPU a cada *pod*, éstos van a tener un peor rendimiento que los que tienen una calidad de servicio *BestEffort*, ya que, en este caso, los *pods BestEffort* cogen la cantidad de CPU que necesitan. Cabe indicar

que, a partir de los 6 contenedores en ejecución de forma simultánea, los tiempos de ejecución empiezan a subir, ya que es el momento en el que el sistema empieza a sobrecargarse.

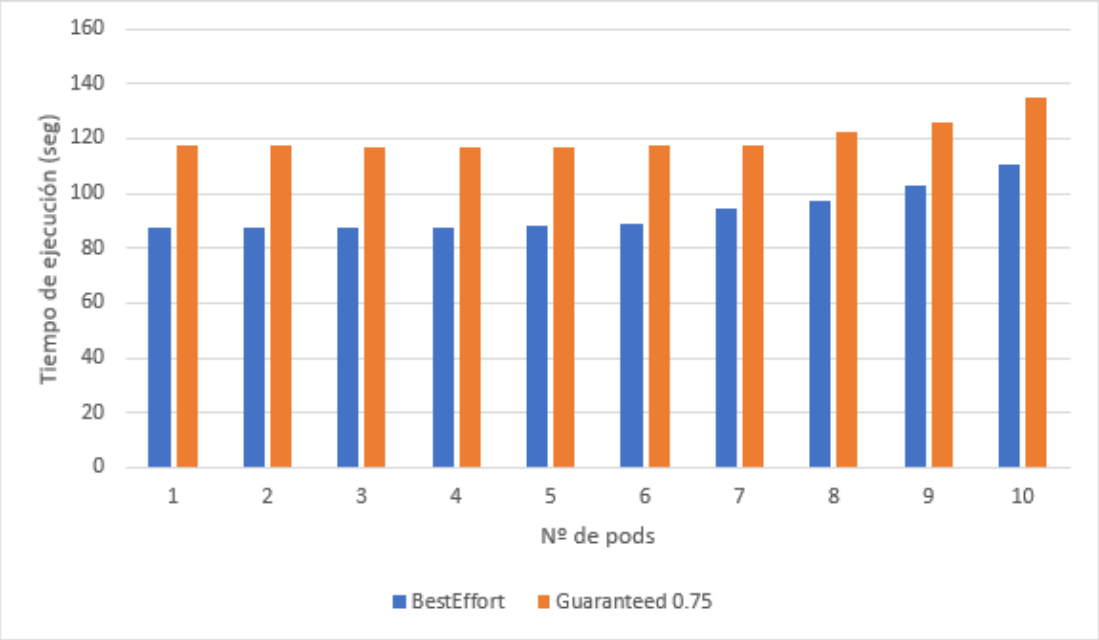

Figura 23: Carga BestEffort vs Guaranteed (CPU=0.75)

En una segunda prueba, el escenario planteado es el mismo que el anterior, pero con la diferencia de que los pods con una calidad de servicio *Guaranteed* tienen 1,25 unidades de CPU, lo que significa que cada pod va a tener algo más de la CPU que necesita por defecto para poder ejecutarse. Como se puede observar en la Figura 24, los *pods BestEffort* tienen la misma problemática que el caso anterior, en donde a partir del momento que el sistema empieza a sobrecargarse, sus tiempos de ejecución empiezan a crecer. Sin embargo, como los *pods Guaranteed* hace mayor uso de la CPU, sus tiempos de ejecución no se ven afectados, ya que usan en todo momento la cantidad de CPU asignada. Entonces, cada *pod* finaliza antes y no se observa degradación en los tiempos de respuesta.

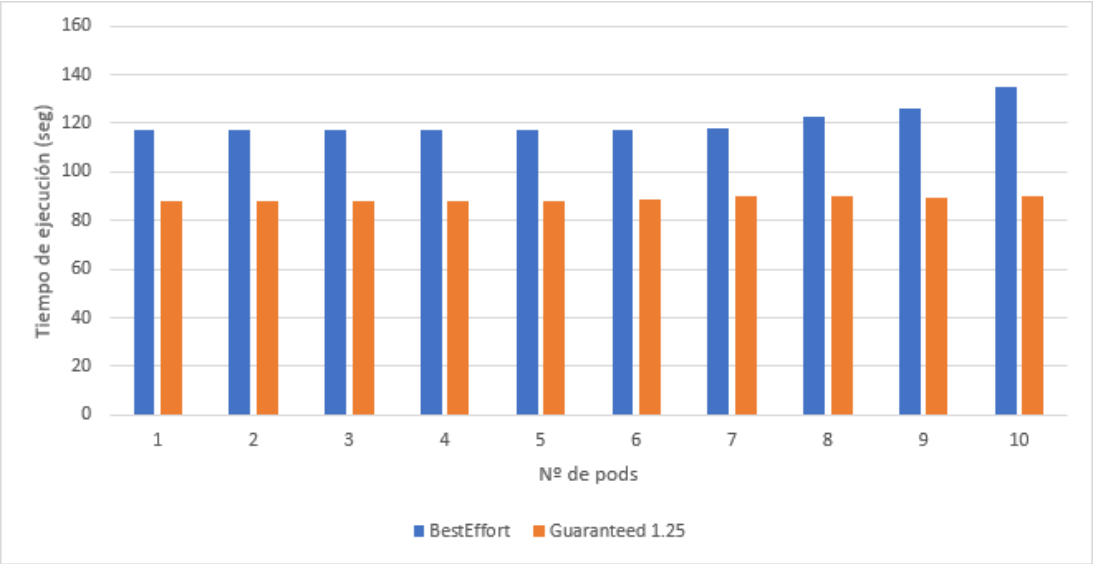

Figura 24: Carga BestEffort vs Guaranteed (CPU=1.25)

Como tercera prueba, el escenario planteado busca realizar una comparativa entre una tanda de *pods* con una calidad de servicio que sea *Guaranteed* con 0,75 unidades de CPU aseguradas y otra tanda de *pods Burstable* los cuales se les aplica un *requests* de 0,75 unidades de CPU. Como se puede observar en la Figura 25, los pods que tienen una calidad de servicio *Burstable* tienen un rendimiento similar a la calidad *BestEffort*, ya que los *pods Burstable* tienen asegurado un mínimo de CPU que es menor que la que coge los *pods* cuando es *BestEffort*.

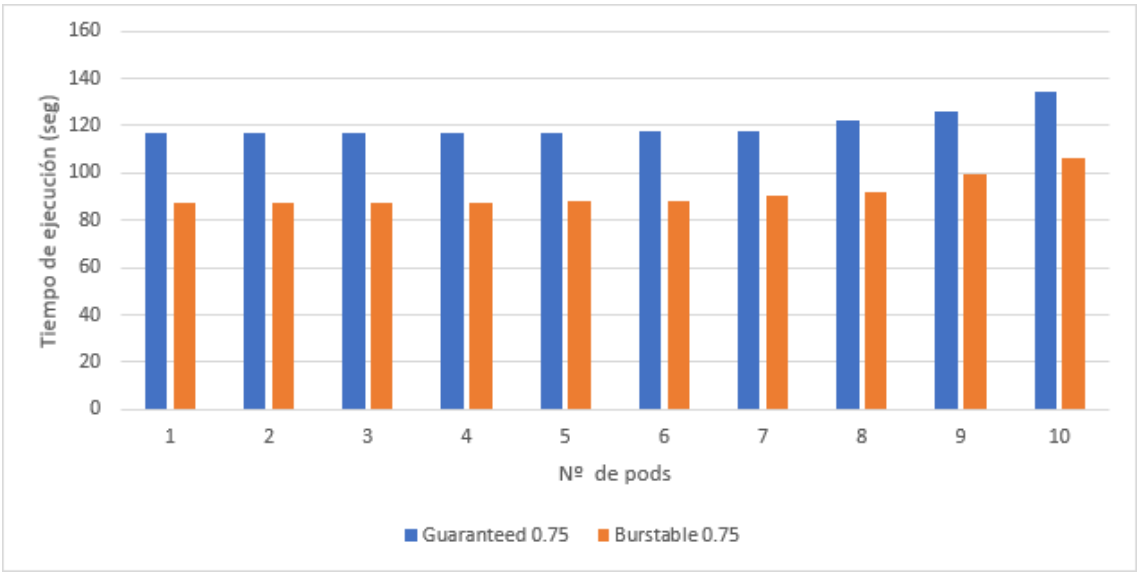

Figura 25: Carga BestEffort vs Burstable (CPU=0.75)

La última prueba realizada dentro de este escenario es igual que la anterior, pero con la diferencia de que en ambas tandas de *pods* asignamos más cantidad de CPU. Como

podemos observar en la Figura 26, las primeras tandas de *pods* tienen tiempos de ejecución similares ya que, con la CPU asignada, el sistema no se sobrecarga, sin embargo, a partir de los 6 contenedores en ejecución, los tiempos de ejecución suben ya que el sistema tiene todos los recursos asignados, y, por tanto, los *pods* necesitan más tiempo para terminar su ejecución. Sin embargo, es una subida en tiempos de ejecución mínima, ya que como se pueda observar en la escala de la gráfica, apenas hay 3 segundos de diferencia entre el más rápido y el más lento, suponiendo un incremento de sólo un 1,04% de diferencia entre *Guaranteed* y *Burstable* en el caso en el que hay una mayor diferencia de tiempos (10 *pods* ejecutándose).

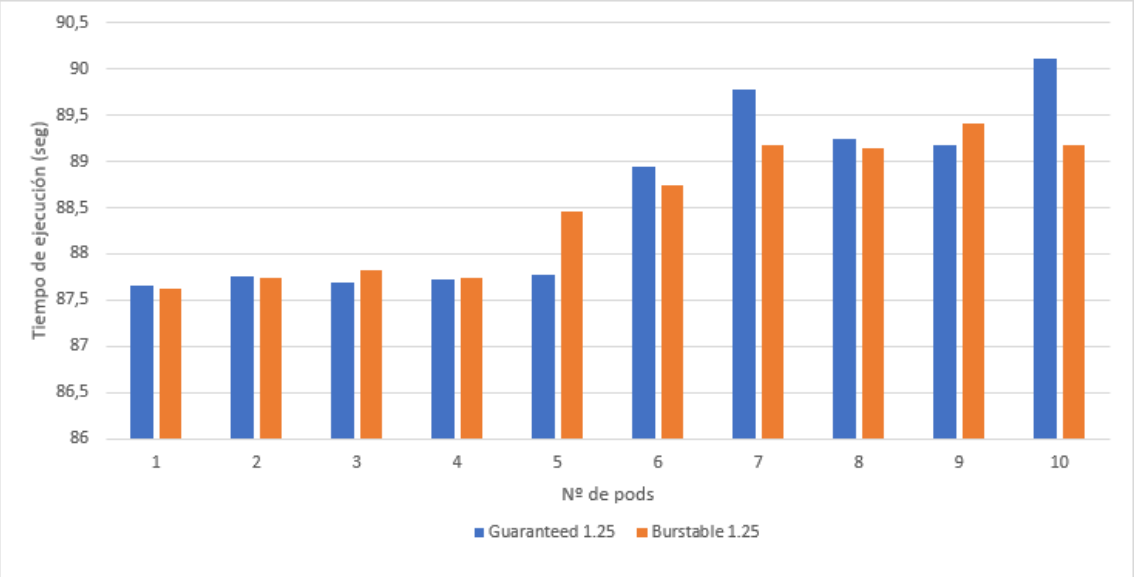

Figura 26: Carga Guaranteed vs Burstable (CPU=1.25 en ambos)

4.2 Resultados del segundo escenario planteado

En este segundo escenario se buscar comprobar cómo actúa el planificador de Kubernetes cuando se cambian las posibles políticas de planificación en cada nodo. En relación a los resultados obtenidos, en la primera prueba se decidió marcar el nodo esclavo *worker-2* con la *taint NoExecute*, la cual evita que ningún nodo sea desplegado sobre dicho nodo. Por tanto, puesto que por defecto el clúster tampoco permite que se despliegen nodos sobre el nodo maestro, al realizar un despliegue de 10 contenedores, todos estos pods son desplegados en el nodo esclavo *worker-1*. Además, como este despliegue tiene una calidad de servicio *Guaranteed* en la que cada *pod* tiene asignadas 1,25 unidades de CPU, como se puede observar en la Figura 27, tenemos la

representación de la ejecución de los *pods* desplegados en el nodo worker-1. Debido a la cantidad de unidades de CPU que requiere cada *pod*, solo pueden ejecutarse 3 a la vez.

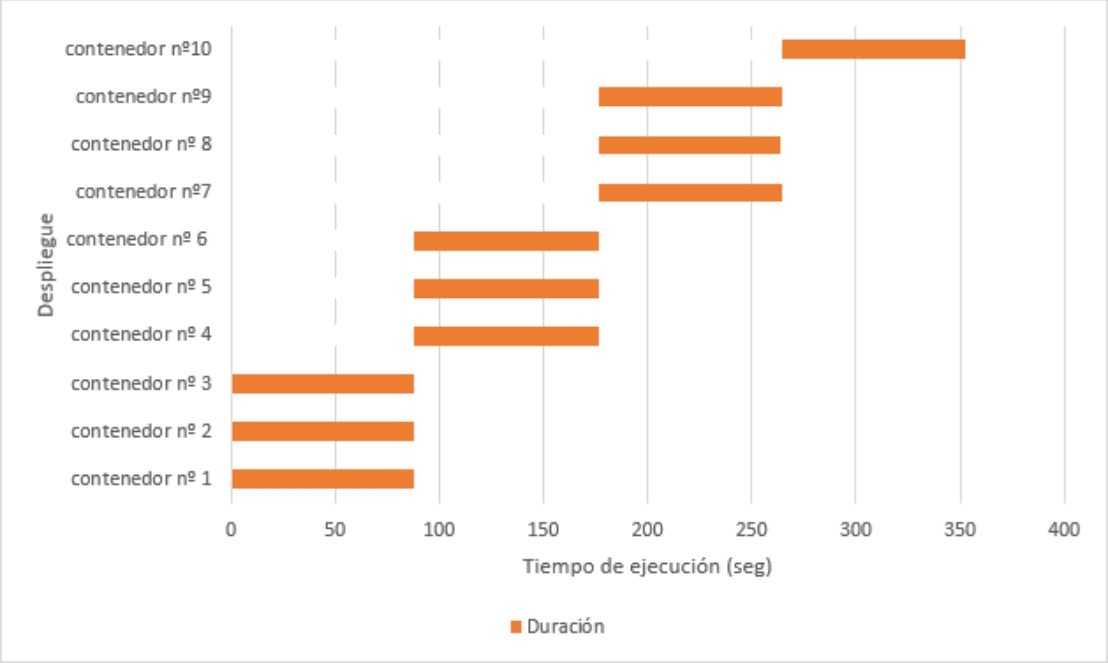

Figura 27: Efecto del parámetro *NoExecute*: Planificación de contenedores con QoS Guaranteed en worker-1

En la segunda prueba realizada, se decidió comprobar el funcionamiento de la *taint PreferNoSchedule,* el cual se puso esta restricción sobre el nodo esclavo *worker-2*. Esta restricción evita desplegar ningún pod sobre el nodo en la que se encuentra si no es necesario. Como podemos observar en la Figura 28, en color naranja tenemos representado el pod que se ejecuta en el nodo que tiene la restricción, mientras que en azul tenemos el resto de pods desplegados en el nodo sin restricción. Al realizar un despliegue de 6 *pods* con una calidad de servicio *Guaranteed* en la que se le asignan 0,75 unidades de CPU a cada pod, puesto que cada nodo cuenta con un máximo de 4 CPUs, al verse el nodo saturado, desvía uno de los *pods* al nodo con la *taint* aplicada.

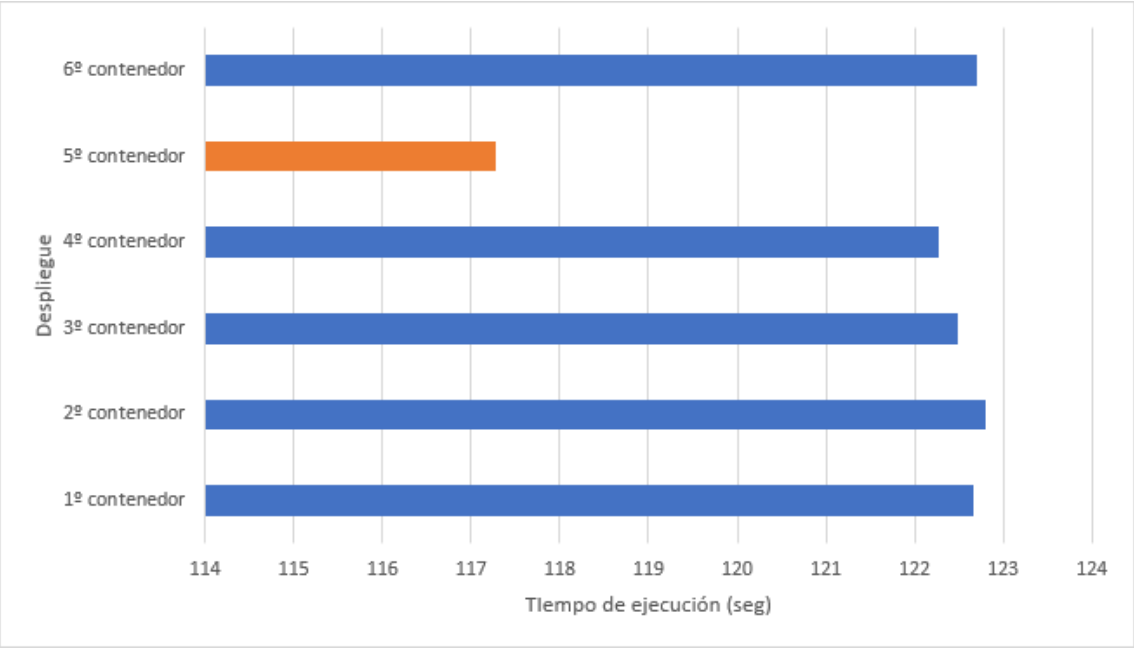

Figura 28: Efecto del parámetro *PreferNoSchedule*. Planificación de contenedores con QoS Guaranteed

En la tercera prueba, se decidió comprobar la misma *taint PreferNoSchedule* pero en el nodo maestro del clúster. Este se nodo se caracteriza por tener un *Hardware* mejor que el de los nodos esclavos. Tras realizar un despliegue de 20 pods con una calidad de servicio *Burstable* en la que cada pod tiene un *requests* de 0,75 unidades de CPU, como podemos observar en la Figura 29, que nos muestra el reparto de los pods en los 3 nodos (en azul el nodo maestro, en verde el nodo worker-1 y en rojo el nodo worker-2) y el tiempo que tardan en ejecutarse, los nodos esclavos se saturan prácticamente al completo, de tal forma que el planificador tiene que desplegar 4 de los *pods* en el nodo maestro. Como el nodo maestro no está saturado, estos *pods* tienen mayor capacidad para poder terminar su ejecución antes, por eso, aunque tenga una política de planificación en la cual es preferible no desplegar *pods,* el planificador se encuentra con la situación en la que no tiene otra opción.

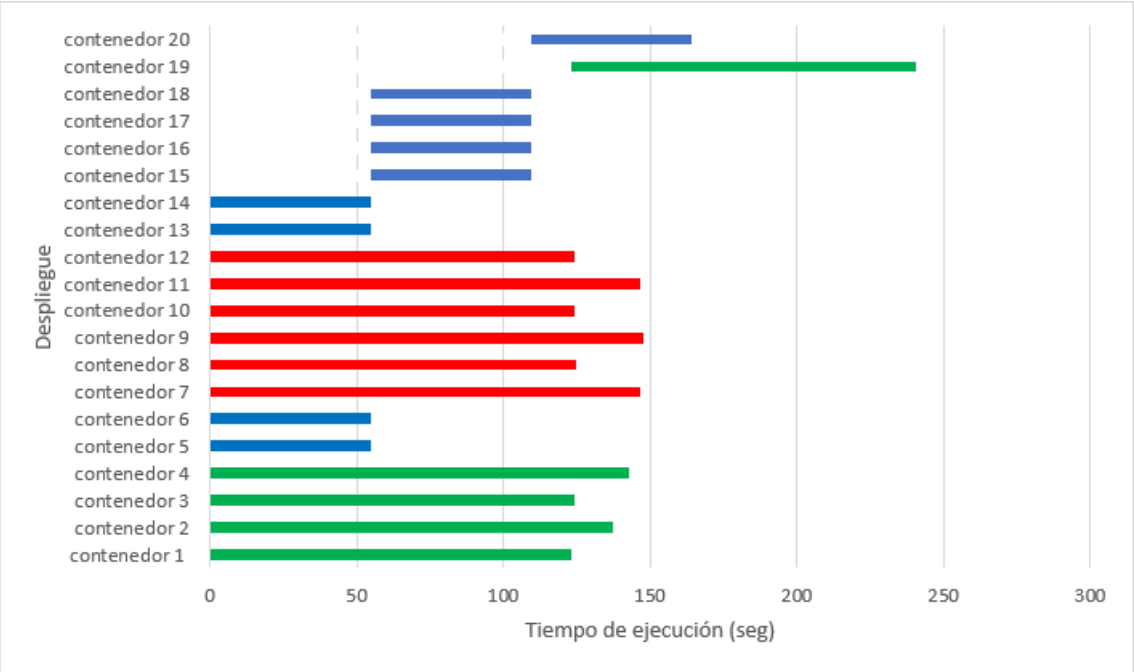

Figura 29: Efecto del parámetro *PreferNoSchedule*. Planificación de contenedores con QoS Burstable (CPU=0.75) y el nodo maestro disponible

En la Figura 30 podemos comprobar cómo es la distribución de *pods* en cada nodo, en donde claramente puede observarse cómo el nodo maestro aun teniendo una *taint* restrictiva, es el nodo que más *pods* ejecuta.

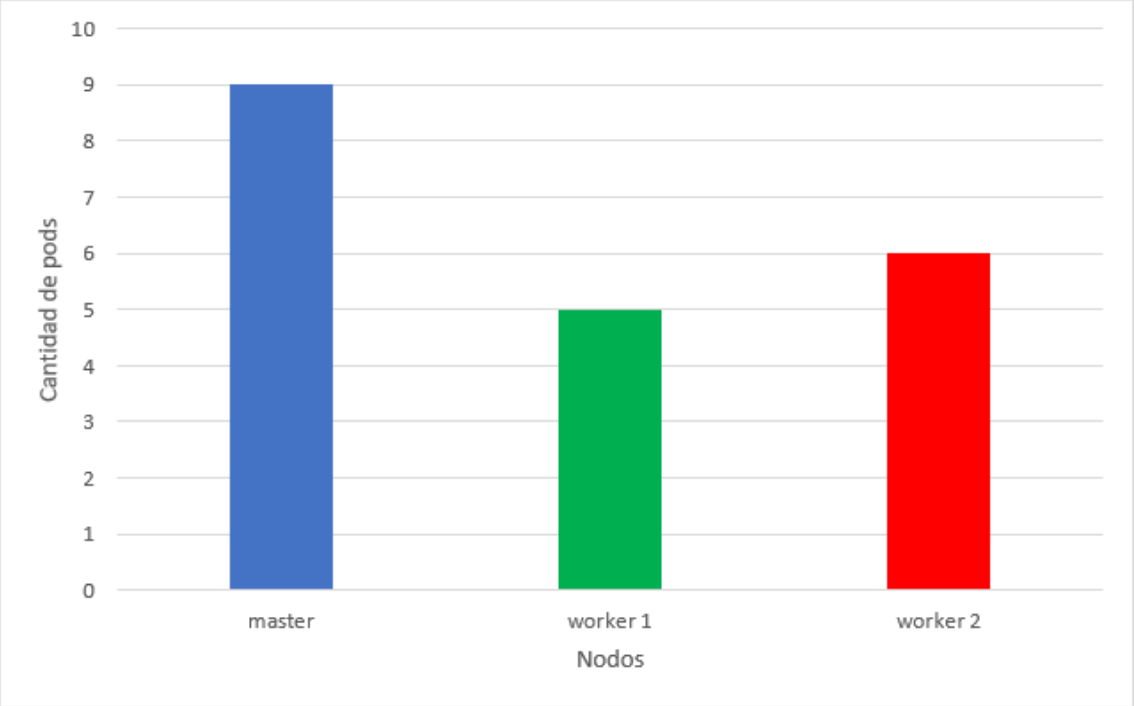

Figura 30: Reparto de pods en la tercera prueba

En la cuarta prueba, el planteamiento es el mismo que el anterior caso, pero con la principal diferencia que cada pod tiene un *requests* de 1,25 unidades de CPU. Como podemos observar en la Figura 31, la situación es similar al caso anterior, los dos nodos esclavos tienen *pods* desplegados hasta que se saturan y el nodo maestro tiene varios *pods* que va ejecutando sólo en los casos en los que detecta que los nodos esclavos no pueden tener más *pods.*

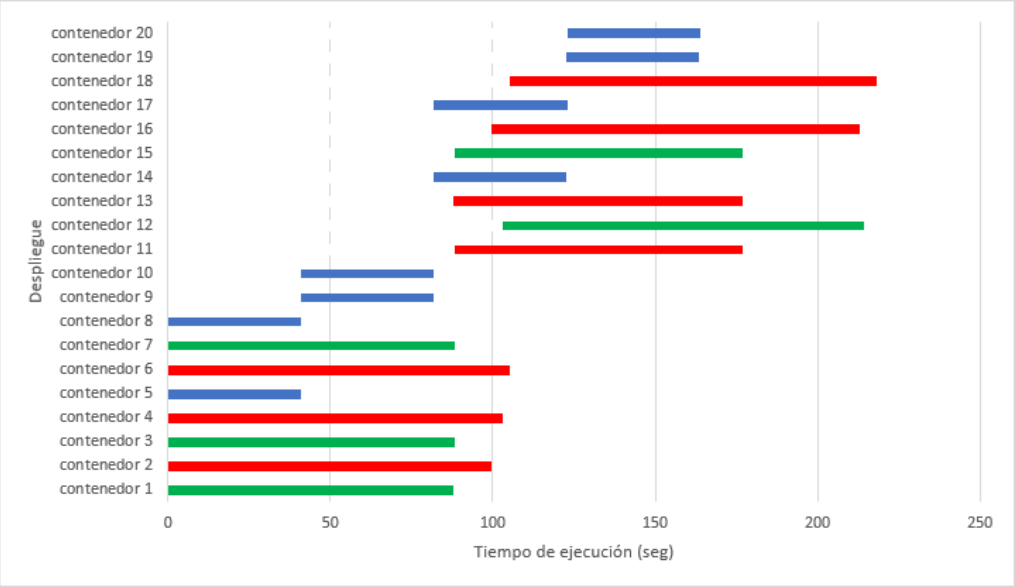

Figura 31: Efecto del parámetro PreferNoSchedule. Planificación de contenedores con QoS Burstable (CPU=1.25) y el nodo maestro disponible

Además, como se puede observar en la Figura 32, el reparto de *pods* en cada nodo es similar al anterior, pero con la diferencia que hay uno más en el nodo esclavo *worker-2* comparándolo con el caso anterior.

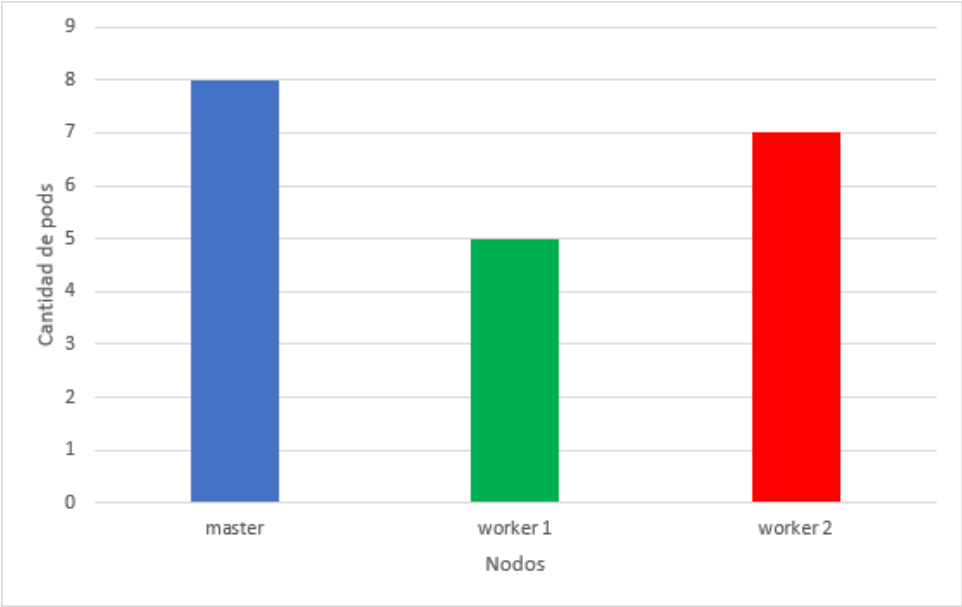

Figura 32: Reparto de *pods* en la cuarta prueba

4.3 Resultados del tercer escenario planteado

En este tercer escenario, se busca comprobar cómo actúa la expropiación de recursos sobre los *pods BestEffort* cuando el sistema está sobrecargado. En relación a los resultados obtenidos, en la primera prueba se propuso comprobar cómo los *pods* que tienen un tipo de calidad de servicio *Guaranteed* expropia recursos a otros *pods* con una calidad de servicio *BestEffort*.

La Figura 33 representa la gráfica en la que se comparan los tiempos de ejecución de 6 *pods* con calidad de servicio *BestEffort* con los tiempos de ejecución de otros 6 *pods* con calidad de servicio *BestEffort* pero con el añadido que se ejecutan a la vez que otros 4 *pods* con una calidad de servicio *Guaranteed* en la que cada *pod* tiene garantizadas 0.75 unidades de CPU. Por ello, al tener en total 3 unidades de CPU reservadas para estos 4 *pods*, los otros 6 *pods* se ven obligados a utilizar sólo una unidad de CPU entre ellos, por lo que su tiempo de ejecución se ve claramente afectado.

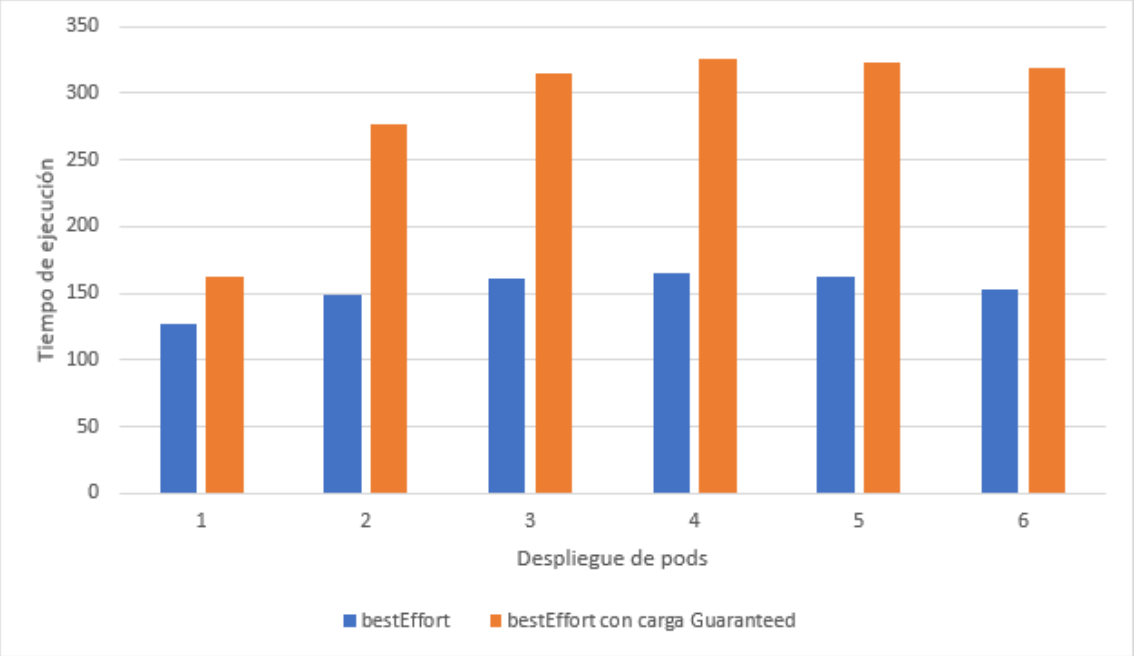

Figura 33: Tiempo de ejecución de *pods BestEffort* y *BestEffort* con carga de *pods Guaranteed* (CPU=0.75)

En la segunda prueba, se propuso comprobar cómo actuarían los tiempos de ejecución de los *pods* si lanzábamos dos tandas de *pods* con una calidad de servicio *Burstable* en ambos, pero en uno se le asignada 0,75 unidades de CPU en *limits* y en la otra tanda se le asigna 0,75 unidades de CPU en *requests*. Como se puede observar en la Figura 34, no

hay muchas sorpresas en cuanto a los resultados obtenidos en esta prueba, ya que los pods que están limitados tienen un rendimiento peor que los que no. Además, no se aprecia que haya ninguna expropiación de CPU ya que todos los *pods* se aseguran que vayan a tener su parte de CPU.

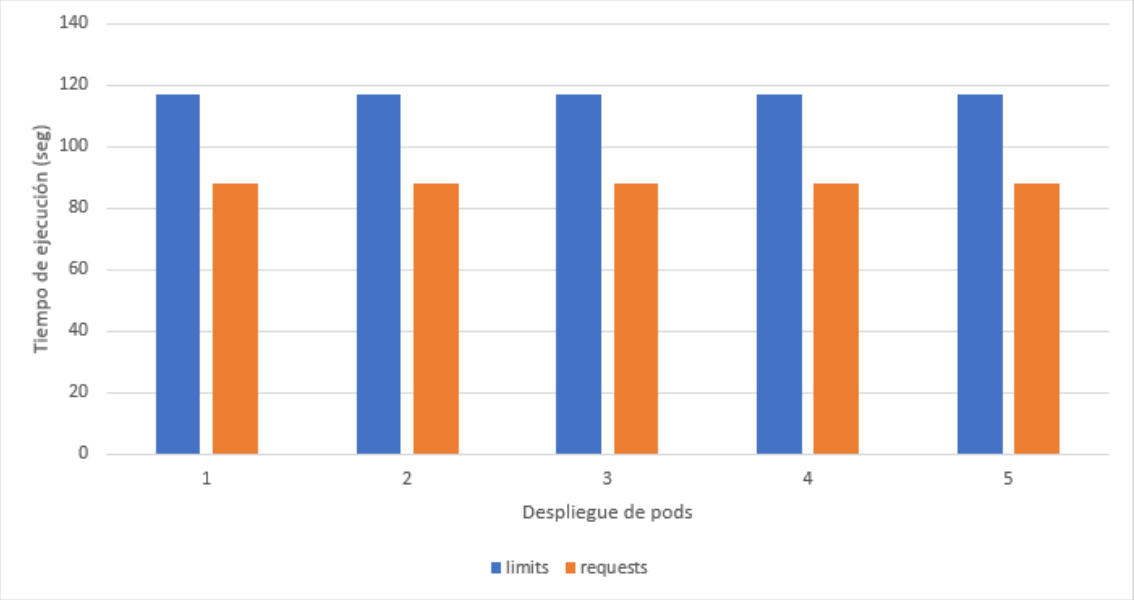

Figura 34: Tiempo de ejecución de *pods Burstable (limit)* y *BestEffort (requests)* con CPU=0.75 en ambos

En la tercera prueba, el escenario propuesto es similar al primero, con la diferencia que vamos a comprobar cómo se ve afectada una tanda de *pods BestEffort* cuando compite en el uso de los recursos con la ejecución de *pods* con una calidad de servicio *Burstable* a la que le asignamos un *requests* de 0,75 unidades de CPU. Como podemos observar en la Figura 33, aunque los *pods* se ven afectados por la expropiación de CPU, ya que tardan más en ejecutarse que los *pods* que no tienen ninguna carga de fondo, la diferencia no es tan grande si lo comparamos con la situación reflejada en la Figura 35, por lo que podemos concluir de que la expropiación realizada cuando los *pods* tienen calidad de servicio *Burstable* no es tan punitiva si la comparamos con la expropiación que realizan los *pods* con calidad de servicio *Guaranteed.*

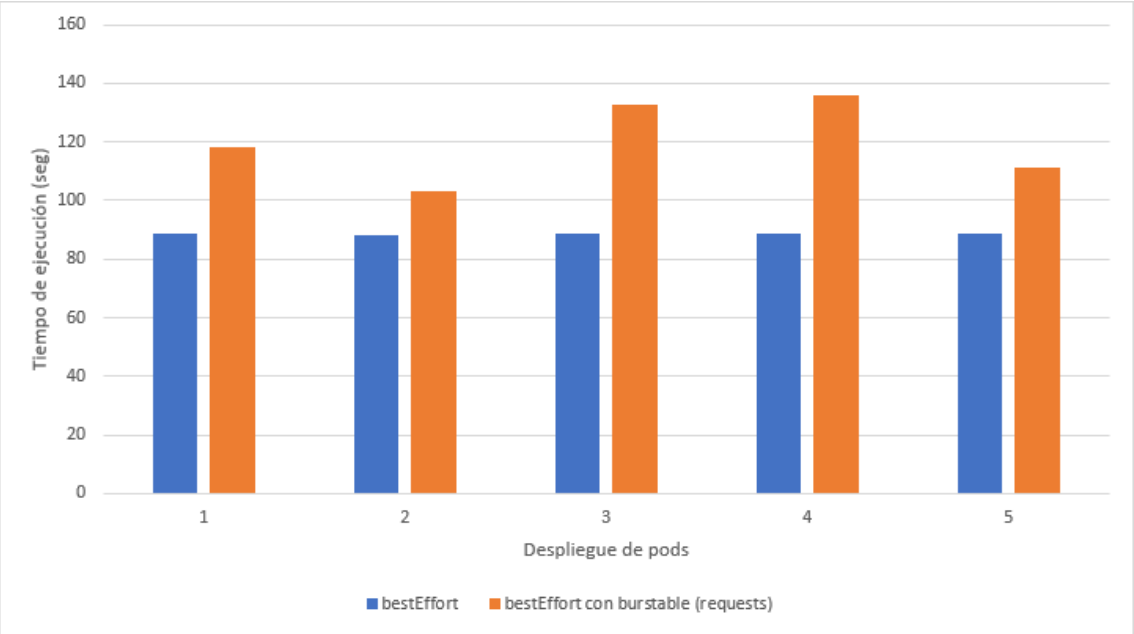

Figura 35: Tiempo de ejecución de *pods BestEffort* y *BestEffort* con carga de *pods Burstable* (CPU=0.75)

4.4 Conclusiones de los resultados

A la luz de estos resultados, es bastante obvia la importancia de una buena planificación a la hora de realizar un despliegue de *pods* en un clúster de Kubernetes. En primer lugar, es importante conocer las distintas opciones de calidad de servicio que nos brinda Kubernetes. Así, las aplicaciones que, por su naturaleza, demanden asegurarse una cierta cantidad de recursos, pueden evitar ser sobrecargadas con ruido de otras aplicaciones.

Cabe señalar la importancia de las restricciones a nivel de nodo, en donde las distintas *taints* nos permiten planificar a los *pods* según la necesidad del momento, y poder descongestionar a los nodos esclavos que estén destinados a la ejecución de aplicaciones desviando la carga de trabajo a otros nodos.

Por último, indicar también la importancia de conocer sobre la expropiación de recursos entre *pods* con distintas calidades de servicio, en donde hemos podido comprobar cómo según se tenga una configuración *Guaranteed* o *Burstable,* el rendimiento en los tiempos de ejecución de algunos *pods* se veía afectado.

Capítulo 5

Conclusiones y Trabajo Futuro

En este capítulo se expondrán las conclusiones e ideas finales obtenidas tras el análisis de los resultados obtenidos en el capítulo anterior. Además, se realizará un esbozo sobre futuras líneas de trabajo y las conclusiones obtenidas por el autor del trabajo.

5.1 Conclusiones

Debido a la expansión del Internet de las Cosas, cada vez nos encontramos con aplicaciones que son desarrolladas pensando en las múltiples posibilidades que surgen en torno a este tipo de paradigma. Sin embargo, estas aplicaciones a veces incluyen ciertos requisitos y restricciones que una arquitectura Cloud tradicional no puede satisfacer. Por ello, surge la computación en la niebla, la cual acerca recursos de computación desde la nube a las "cosas". Además, las técnicas de virtualización basadas en contenedores y los ordenadores monoplaca como las Raspberry Pi son buenos elementos para utilizar en este tipo de arquitecturas.

Durante el desarrollo de este TFG se ha llevado a cabo una evaluación de las distintas posibilidades de planificación de *pods* dentro de un clúster de Kubernetes, en donde hemos podido comprobar en distintos escenarios la importancia de conocer cuántos recursos son asignados a cada *pod*, la importancia de las *taints* a nivel de nodo y cómo el planificador evitar desplegar *pods* en nodos que tengan algún tipo de restricción. También se ha podido comprobar cómo los *pods* que sean desplegados con una calidad

de servicio mayor se encargan de realizar la expropiación de la CPU de otros nodos cuya prioridad de ejecución sea menor.

5.2 Competencias adquiridas

Durante el transcurso de este TFG, se han aplicado las siguientes competencias específicas de la intensificación de Ingeniería de Computadores:

- [IC5] Capacidad de analizar, evaluar, y seleccionar las plataformas hardware y software más adecuadas para el soporte de aplicaciones empotradas y tiempo real.
- [IC7] Capacidad para analizar, evaluar, seleccionar y configurar plataformas hardware para el desarrollo y ejecución de aplicaciones y servicios informáticos.

5.3 Trabajo futuro

Tras el trabajo realizado en este TFG, una posible línea de estudio sería desarrollar desde cero un planificador propio para el clúster de Kubernetes, el cual fuese optimizado para una aplicación concreta. Para ello, sería necesario el estudio del lenguaje de programación correspondiente, y un estudio en profundidad del funcionamiento de la API de Kubernetes.

Además, junto a la creación de este planificador, podría probarse en un sistema real en donde el análisis no sólo se centre en la capa de computación en la niebla, si no que dicho análisis se expanda también a la computación en la nube. Así, se podría realizar un estudio sobre un entorno real en el que se midan tanto el comportamiento de un planificador personalizado, como el rendimiento de las latencias de red y así, comprobar que el sistema de computación en la niebla genera una mejor respuesta en sistemas de tiempo real.

5.4 Conclusiones personales

A nivel personal, estoy satisfecho con el trabajo realizado en el transcurso de este TFG. Ha sido mi primer contacto en profundidad con una tecnología de orquestación de contenedores como es Kubernetes, por lo que estoy convencido de que le daré uso a todos los conocimientos obtenidos en el desarrollo del estudio de dicha tecnología, ya que en el ámbito laboral cada vez está siendo más común su uso. En lo que respecta a

los resultados obtenidos, creo que refleja lo importante que es una buena planificación de cara a obtener un clúster cuya ejecución sea la más óptima posible.

Bibliografía

[1] Atlam, H.F.; Walters, R.J.; Wills, G.B. Fog Computing and the Internet of Things: A Review. *Big Data Cogn. Comput.* 2018, *2*, 10. https://doi.org/10.3390/bdcc2020010. - 1, 5, 6, 7, 8, 9

[2] W. Yu *et al.*, "A Survey on the Edge Computing for the Internet of Things," in *IEEE Access*, vol. 6, pp. 6900-6919, 2018, doi: 10.1109/ACCESS.2017.2778504. - 1

[3] A.V. Dastjerdi, H. Gupta, R.N. Calheiros, S.K. Ghosh, R. Buyya, Chapter 4 - Fog Computing: principles, architectures, and applications, Editor(s): Rajkumar Buyya, Amir Dastjerdi, Internet of Things, Morgan Kaufmann, 2016, Pages 61-75, ISBN 9780128053959, https://doi.org/10.1016/B978-0-12-805395-9.00004-6. - 7, 10, 11

[4] ¿Qué es la virtualización? Funciones y seguridad de la virtualización. https://www.redhat.com/es/topics/virtualization/what-is-virtualization - Accedido en junio de 2021. - 14

[5] ¿Qué es un hipervisor? - https://www.redhat.com/es/topics/virtualization/what-is-a-hypervisor - Accedido en junio de 2021 - 14

[6] Tecnologías de virtualización basada en contenedores – https://blog.redigit.es/tecnologias-de-virtualizacion-basada-en-contenedores/ - Accedido en junio de 2021 – 14, 15

[7] Contenedores de software - ¿Qué es Docker? - https://www.redhat.com/es/topics/containers/what-is-docker - Accedido en junio de 2021 – 16, 17

[8] Qué es una placa SBC o Single Board Computer - https://descubrearduino.com/sbc/ - Accedido en junio de 2021 – 11, 12

[9] ¿Qué es una Raspberry Pi? - https://descubrearduino.com/breve-guia-de-la-raspberry-pi/ - Accedido en junio de 2021 - 13

[10] 15 usos de la Raspberry Pi que no sabías que podías darle - https://computerhoy.com/noticias/hardware/15-usos-raspberry-pi-que-no-sabias-que-podias-darle-74905 - Accedido en junio de 2021 - 13

[11] Las 10 herramientas más importantes más importantes para orquestación de contenedores Docker - https://www.campusmvp.es/recursos/post/las-10-herramientas-mas-importantes-para-orquestacion-de-contenedores-docker.aspx - Accedido en junio de 2021. - 22

[12] ¿Qué es Kubernetes? - https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/ - Accedido en junio de 2021. - 1, 19

[13] Punto de acceso WiFi con Raspberry Pi - https://hipertextual.com/2020/07/punto-acceso-wifi-raspberry-pi - Accedido en junio de 2021 - 12

[14] Entender los objetos de Kubernetes - https://kubernetes.io/es/docs/concepts/overview/working-with-objects/kubernetes-objects/ - Accedido en junio de 2021. – 20, 21

[15] ¿Qué es un pod de Kubernetes? - https://www.redhat.com/es/topics/containers/what-is-kubernetes-pod - Accedido en junio de 2021. - 19

[16] Kubernetes Scheduler - https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/ - Accedido en junio de 2021. - 2, 22

[17] A Brief Analysis on the Implementation of the Kubernetes Scheduler - https://www.alibabacloud.com/blog/a-brief-analysis-on-the-implementation-of-the-kubernetes-scheduler_595083 - Accedido en junio de 2021. - 23

[18] Scheduling Framework - https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/ - Accedido en junio de 2021. - 24

[19] Writing custom Kubernetes schedulers - https://banzaicloud.com/blog/k8s-custom-scheduler/ - Accedido en junio de 2021. - 24

[20] Controla la programación con taints de nodo - https://cloud.google.com/kubernetes-engine/docs/how-to/node-taints?hl=es-419 - Accedido en junio de 2021. - 24

[21] Scheduling policies - https://kubernetes.io/docs/reference/scheduling/policies/ - Accedido en junio de 2021. - 25

[22] Administrar los recursos para tus contenedores de Kubernetes - https://www.returngis.net/2020/05/administrar-los-recursos-para-tus-contenedores-en-kubernetes/ - Accedido en junio de 2021. - 26

[23] Kubernetes best practices: Resource resquests and limits - https://cloud.google.com/blog/products/containers-kubernetes/kubernetes-best-practices-resource-requests-and-limits - Accedido en junio de 2021. – 25, 26

[24] P. Kayal, "Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope : Invited Paper," 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), 2020, pp. 1-6, doi: 10.1109/WF-IoT48130.2020.9221340. - 5

[25] Gestión inteligente del tráfico en las Smart Cities - https://theconversation.com/gestion-inteligente-del-trafico-en-las-smart-cities-109530 - Accedido en junio de 2021. - 6

[26] Hacia el Fog Computing, ¿En qué punto estamos? - https://iotfutura.com/2019/06/hacia-el-fog-computing-en-que-punto-estamos/#Gestion_del_agua – Accedido en junio de 2021. - 6

[27] 14 herramientas de orquestación de contenedores para DevOps - https://geekflare.com/es/container-orchestration-software/ - Accedido en junio de 2021. - 14

[28] Kubernetes - https://insujang.github.io/2019-11-07/kubernetes/ - Accedido en junio de 2021. - 19

[29] Viewing Pods and Nodes - https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/ - Accedido en junio de 2021. - 19

[30] ReplicaSet - https://kubernetes.io/es/docs/concepts/workloads/controllers/replicaset/ - Accedido en junio de 2021. - 21

[31] Deployment - https://kubernetes.io/es/docs/concepts/workloads/controllers/deployment/ - Accedido en junio de 2021. - 21

[32] StatefulSets - https://kubernetes.io/es/docs/concepts/workloads/controllers/statefulset/ - Accedido en junio de 2021. - 21

[33] Raspberry Pi: benchmark de sistema con Sysbench - https://protegermipc.net/2017/05/31/raspberry-pi-benchmark-de-sistema-con-sysbench/ - Accedido en junio de 2021. – 30, 31

[34] Imagen de Sysbench para ARM - https://hub.docker.com/repository/docker/angel96eur/sysbench - Accedido en junio de 2021. - 32

[35] Writing custom Kubernetes schedulers - https://banzaicloud.com/blog/k8s-custom-scheduler/ - Accedido en junio de 2021. - 36

[36] Random Scheduler - https://github.com/martonsereg/random-scheduler - Accedido en junio de 2021. – 36, 37

[37] Raspberry Pi 3 Model B - https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ - Accedido en junio de 2021. - 28

[38] Raspberry Pi4 Model B - https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/ - Accedido en junio de 2021. - 28

[39] Yu Cao, Songqing Chen, Peng Hou and D. Brown, "FAST: A fog computing assisted distributed analytics system to monitor fall for stroke mitigation," 2015 IEEE International Conference on Networking, Architecture and Storage (NAS), 2015, pp. 2-11, doi: 10.1109/NAS.2015.7255196. - 10

[40] Nomad vs Kubernetes without the complesy - https://www.codemotion.com/magazine/dev-hub/backend-dev/nomad-kubernetes-but-without-the-complexity/ - Accedido en junio de 2021. - 18

[41] Docker Compose y Docker Swarm: orquestación de contenedores Docker - https://www.ionos.es/digitalguide/servidores/know-how/docker-compose-y-swarm-gestion-multicontenedor/ - Accedido en junio de 2021. - 18

[42] Conceptos de Kubernetes - https://kubernetes.io/es/docs/concepts/ - Accedido en junio de 2021. - 20

[43] kube-apiserver - https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/ - Accedido en junio de 2021. - 20

[44] kube-controller-manager - https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/ - Accedido en junio de 2021. - 20

[45] kube-scheduler - https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/ - Accedido en junio de 2021. - 20

[46] kubelet - https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ - Accedido en junio de 2021. - 20

[47] Kube-proxy - https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/ - Accedido en junio de 2021. - 20

[48] Componentes de Kubernetes - https://kubernetes.io/es/docs/concepts/overview/components/ - Accedido en junio de 2021. - 20

[49] Espacios de nombres - https://kubernetes.io/es/docs/concepts/overview/working-with-objects/namespaces/ - Accedido en junio de 2021. - 20

[50] Despliegue de aplicaciones con Docker Compose - https://colaboratorio.net/davidochobits/sysadmin/2018/despliegue-de-aplicaciones-con-docker-compose/ - Accedido en junio de 2021. - 18

[51] Orquestación con Kubernetes - https://www.itdo.com/blog/orquestacion-con-kubernetes/ - Accedido en junio de 2021. - 22

[52] Google Kubernetes Engine - https://cloud.google.com/kubernetes-engine#section-5 – Accedido en junio de 2021. - 22

[53] Azure Kubernetes Service (AKS) - https://azure.microsoft.com/es-es/services/kubernetes-service/#features – Accedido en junio de 2021. - 22

[54] Microsoft Teams - https://www.microsoft.com/es-es/microsoft-teams/group-chat-software - Accedido en Julio de 2021. - 3

[55] Metodología SCRUM - https://www.wearemarketing.com/es/blog/metodologia-scrum-que-es-y-como-funciona.html - Accedido en Julio de 2021. - 3

Anexo I. Scripts

I.1 Script de instalación

I.1.1. Script básico para todos los nodos

```
sudo apt update && sudo apt dist-upgrade

cat /boot/cmdline.txt >>
  cgroup_enable=cpuset cgroup_memory=1 cgroup_enable=memory

sudo dphys-swapfile swapoff
sudo dphys-swapfile unistall
sudo apt purge dphys-swapfile

# Instalación de Docker
curl -sSL get.docker.com | sh
sudo usermod -aG docker pi

# Añadimos el repositorio de Kubernetes y realizamos su
#instalación
sudo vim /etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg
| sudo apt-key add –
sudo apt update
sudo apt install kubeadm kubectl kubelet
```

I.1.2. Script para configurar el nodo maestro

```
# comando para que el nodo maestro inicie el cluster
sudo kubeadm init --pod-network-cidr=10.244.0.0/16
mkdir -p ~.kube
sudo cp /etc/kubernetes/admin.conf ~/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Docum
entation/kube-flannel.yml
```

I.1.3. Script para configurar cada nodo en el clúster

```
# comando para unir un nodo al cluster
kubeadm join 192.168.1.15:6443 --token 4zrfhy.49o3wtix9jd0n29k
\
    --discovery-token-ca-cert-hash
sha256:05a5c82e83062e43366112d5b8be732e23346a0ea723acef4e8f802
576c4a99a
```

I.2 Script para desplegar pods

```
#!/bin/bash

# $1 es el archivo yaml del despliegue, $2 la cantidad de
replicas que queremos, $3 es el numero primo a calcular
echo "despliegue: $1"
echo "cantidad de pods: $2"
echo "numero primo: $3"
echo "valor de sleep: $4"

# Hacemos una copia del ejecutable inicial para no modificarlo
cp $1 copia.yaml

# Sustituimos la x en los lugares clave para que podamos tener
# varios despliegues distintos

for ((i = 0 ; i < $2 ; i++)); do
  # Cogemos el la hora a la que se empieza a ejecutar el pod
  echo $(date +%T) >> fecha
  sed -i 's/:/-/g' fecha
  tiempo=$(cat fecha)
  rm fecha

  # Adecuamos el fichero a desplegar
        sed -i 's/q/'$3'/g' copia.yaml
  sed -i 's/tiempo/'$tiempo'/g' copia.yaml
        sed -i 's/sleep/'$4'/g' copia.yaml

  # Ejecutamos el despliegue
  kubectl apply -f copia.yaml
        sed -i 's/'$3'/q/g' copia.yaml
  sed -i 's/'$tiempo'/tiempo/g' copia.yaml
        sed -i 's/'$4'/sleep/g' copia.yaml
  sleep $4
done
rm copia.yaml
```

I.3 Script para recoger datos

```
#!/bin/bash

# Guardar resultados en un archivo
kubectl get pods | tee salida.txt

# Miro los que estan completos y los añado
awk '$3=="Completed" {print $1}' salida.txt >> salida1.txt
rm salida.txt

# Cuento los que hay
tambucle=$(cat salida1.txt | wc -l)

# Genero un bucle con tantas iteraciones como lineas tenga
# salida1.txt,
# cojo la primera linea que es el nombre de pod, compruebo su
# log y guardo
# el resultado en un archivo. Borro esa primera linea y
# vuelvo al principio del bucle otra vez
for (( c=0; c<tambucle; c++ )); do
        variable=$(head -1 salida1.txt)
# Cogemos la primera linea del archivo que contiene los pods
# que han sido completados

        echo $variable >> resultado.txt
        kubectl logs $variable >> resultado.txt
# guardamos la salida en el archivo resultado.txt

  kubectl describe pod $variable >> variable.txt
  kubectl delete pod $variable
        awk '$1=="Node:" {print}' variable.txt >>
resultado.txt

# Guardamos en que nodo estaba el pod actual
        rm variable.txt
        tail -n +2 salida1.txt >> salidaAux.txt
# borramos la primera linea
        rm salida1.txt
        cat salidaAux.txt >> salida1.txt
        rm salidaAux.txt
        echo "-----" >> resultado.txt
done
```

```
# Buscamos las lineas que indican el nodo, el nombre del pod y
su tiempo

grep -i -E "Node:|sysbench-|total time:" resultado.txt >>
resultados1.txt

awk '{if($1 ~ /s[sysbench-]/){print $1;}if($1=="total"){print
$3;}if($1=="Node:"){print $2;}}' resultados1.txt >>
resultados.txt

rm resultado.txt resultados1.txt salida1.txt

#Arreglamos el archivo para obtener un archivo csv
sed -e 's/s$//' resultados.txt >> resultados1.txt
#Quitamos la s de los tiempo

echo "Nombre;Tiempo;Nodo" >> resultados.csv

tam=$(cat resultados1.txt | wc -l)

while [ $tam -gt 0 ]
do
        parte1=$(sed -n '1p' resultados1.txt)
        parte2=$(sed -n '2p' resultados1.txt)
        parte3=$(sed -n '3p' resultados1.txt)
        parte="${parte1};${parte2};${parte3}"
        echo $parte >> resultados.csv

        for i in 1 2 3
        do
                sed -i -e "1d" resultados1.txt
        done

        tam=$(cat resultados1.txt | wc -l)

done

echo " " >> resultados.csv

rm resultados1.txt resultados.txt
```

Anexo II. Archivos YAML

II.1 YAML usado en la ejecución de pods

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: sysbench
  name: sysbench-q-tiempo-sleep
spec:
  containers:
  - command:
    - sysbench
    - --test=cpu
    - --cpu-max-prime=q
    - run
    image: angel96eur/sysbench
    name: sysbench
  restartPolicy: Never
```

II.2 YAML usado en la ejecución de pods con limits y requests

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: sysbench
  name: sysbench-q-sleep
spec:
  containers:
  - command:
    - sysbench
    - --test=cpu
    - --cpu-max-prime=q
    - run
    image: angel96eur/sysbench
    name: sysbench
  resources:
    limits:
      cpu: " "
      memory: " "
  restartPolicy: Never
```

II.3 YAML con un despliegue de nginx

```
apiVersion: apps/v1 # Usa apps/v1beta2 para versiones
anteriores a 1.9.0
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # indica al controlador que ejecute 2 pods
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Anexo III. Dockerfiles

III.1 Dockerfile para la imagen de Sysbench en ARM

```
FROM debian:latest

RUN apt-get update && \
        apt-get -y install curl && \
        apt-get install -y apt-transport-https

RUN printf "deb http://archive.debian.org/debian/ jessie
main\ndeb-src http://archive.debian.org/debian/ jessie
main\ndeb http://security.debian$
RUN apt-get autoclean && \
        apt-get -f install && \
            dpkg --configure -a && \
               apt-get update
RUN apt-get -y -f install sysbench
```