ENTENDIENDO QUÉ ES Y CÓMO FUNCIONA KUBERNETES

RESUMEN

En este trabajo hablamos de Kubernetes, un sistema de orquestación de contenedores de código abierto desde 2014 (cuando fue lanzado por Google) y con una integración especialmente buena con contenedores Docker. Para comprender cuál es exactamente la función que desempeña vamos a ver cuál es la arquitectura de la plataforma, las diversas maneras en las que podemos acceder como administrador y centrarnos en dos conceptos fundamentales: el autoescalado (*autoscaling*) y el balanceo de carga (*load balancing*). Para acabar, miramos cuál es el impacto que ha tenido este sistema en la actualidad, tanto a nivel empresarial como de la comunidad de código abierto y daremos una conclusión de lo que hemos aprendido tras realizar el trabajo.

1. Introducción

Kubernetes "es una plataforma de código abierto para automatizar el despliegue, escalado y operaciones sobre contenedores de aplicaciones entre clústeres de hosts" [1] que busca cumplir los principios de ser portable (utilizando en pequeños y grandes servidores, de manera privada o pública), extensible (el administrador del servidor puede añadir nuevos módulos y/o modificar el código) y "auto curable" (es decir, que sea capaz de responder ante situaciones que puedan llegar a dar un error y prevenirlas para poder garantizar un servicio continuo a los clientes). Este proyecto fue iniciado por Google y donado a la CNCF (*Cloud Native Computing Foundation).* Un contenedor es una herramienta que permite a administradores de sistemas y programadores comprobar el correcto funcionamiento de una aplicación en un entorno igual al de producción pero seguro, de una manera "similar a una máquina virtual pero sin separar el kernel ni emular todo el hardware" [2].

2. Componentes de Kubernetes

Antes de empezar a ver como se encuentra organizada la arquitectura de Kubernetes es necesario conocer algunos conceptos de vocabulario propios al mismo.

Un **pod** [3] es un grupo de uno o más contenedores al que se encuentra asociada una dirección IP (*Internet Protocol)* y el mismo puerto. Los pods son unidades indivisibles a la hora de la planificación y asignación de espacio en disco, así pues, los contenedores que se encuentra en un pod pueden comunicarse fácilmente a través de localhost, o de otros sistemas de comunicación como la memoria compartida POSIX. Los pods son especialmente útiles cuando existen dependencias entre varios contenedores distintos, un claro ejemplo de esto sería ejecutar un servidor LAMP (*Linux Apache MySQL PHP*), en el que todos los elementos dependientes se ejecutarían al iniciar el pod consiguiendo así una planificación conjunta y que si alguno de los elementos falla el pod no llegue a ejecutarse. Así mismo, al eliminar el pod, se eliminarán todos los contenedores asociados al mismo.

Labels Selectors (*selectores de etiquetas)* [4] es un mecanismo utilizado por Kubernetes para seleccionar y organizar subconjuntos de objetos. Esto se consigue gracias a las etiquetas. Una etiqueta es un par "clave-valor" que se adjuntan a objetos (como los pods) y que se utilizan para especificar atributos orientados a la identificación de los mismos. Cada objeto puede tener varias etiquetas asociadas aunque cada clave debe de ser única para un objeto dado. Esto no debe afectar a la estructura del sistema, ya que existen simplemente para facilitar el trabajo a los usuarios que lo utilicen.

Un **controlador de replicación** (*Replication Controller*) [5] se encarga de que haya una serie de copia de un mismo pod, llamadas réplicas, siempre presentes en el sistema, consiguiendo así que el pod o conjunto de pods del que se encargue estén siempre disponibles. Si hubiese muy pocos pods disponibles, el controlador creará más, en caso contrario, es decir, si hubiera demasiados pods disponibles, el controlador se encargará de eliminarlos automáticamente. Además de para asegurar que existan copias del pod ejecutándose en caso de que ocurra algún error puntual, facilita el autoescalado del sistema ya sea manualmente (reconfigurando el número de réplicas) o automáticamente gracias al agente automático de escalado que

Kubernetes proporciona (del que hablaremos más adelante). En caso de que queramos que los pods finalicen por sí sólos en vez tener que ser finalizados a través del controlador se utilizan trabajos (*jobs*) y para utilidades a nivel de máquina como monitorización o "*logging*" se utilizan demonios (*DaemonSet*).

Una ligera mejora sobre el controlador de replicación son los **ReplicaSets**[6]. Mientras que los controladores de replicación utilizan selectores de etiquetas *(label selectors)* basados en igualdades, un ReplicaSet puede aplicar el requisito a un conjunto de valores diferentes. Así pues todo lo que hace un controlador de replicación es realizado por el ReplicaSet aunque en la mayoría de casos no existe una diferencia entre ambos.

Una herramienta similar pero de más alto nivel son los Deployments. Un **Deployment**[7] nos permite realizar todo lo que podíamos hacer con un ReplicaSet, además, podemos declarar una manera en la que se actualizará dicho ReplicaSet. La configuración del Deployment es normalmente realizada mediante un archivo, que se puede modificar a lo largo del tiempo. Además, los deployments guardan un historial de revisiones (limitado por un número de revisiones configurable) que nos permitirá volver a versiones anteriores en caso de que la nueva versión que hayamos creado no cumpla los objetivos deseados.

Como ya se ha explicado antes, los pods tienen un periodo de vida finito en el sistema. El principal problema es que algunos pods pueden proporcionar algún tipo de funcionalidad a otro. Esto es un inconveniente debido a que no se puede confiar ni siquiera en que las direcciones IP son estables en el tiempo, por lo que un pod no tiene forma de saber qué pods tiene que tener activos para seguir funcionando. Un **servicio** [8] en Kubernetes es una abstracción que permite definir un agrupamiento lógico de pods y las políticas para acceder a ellos. El conjunto de pods agrupados mediante un servicio suele venir identificado con la ayuda de las etiquetas. Esto nos permite establecer las dependencias entre los pods que pertenezcan a un mismo servicio, pudiendo permitir incluso disociar pods momentáneamente mientras que las dependencias de funcionalidad no sean violadas.

2. Arquitectura de Kubernetes

Kubernetes sigue una estructura maestro-esclavo. [9] Existe un nodo maestro o de control que se encarga de administrar el resto de nodos de la plataforma Kubernetes.

2.1 El nodo maestro o nodo de control

El nodo maestro sirve para controlar el resto de nodos y por tanto contiene la información necesaria para ello. Podemos acceder a ella a través de **API Server,** los valores almacenados se encontrarán en **etcd**, habrá un **administrador de control** encargado de asegurarse de que dichos valores de configuración se estén cumpliendo y un **planificador** encargado de determinar si un pod puede ir a algún nodo, y en caso afirmativo, escoger el mejor nodo siguiendo la configuración de prioridad que deseemos.

La principal funcionalidad de **API Server** *(Application Programming Interface Server)* es configurar los datos de los objetos de tipo "api" que incluyen los pods, los servicios, controladores de replicación y demás. Principalmente, son un conjunto de operaciones REST *(Representational State Transfer)* y proporciona todos los objetos que necesiten de la funcionalidad de otros al estado de clústeres compartidos con el que el resto de componentes pueden interactuar entre sí[10]. El principal objetivo de esta API es permitir que el sistema crezca a medida que vayan surgiendo la necesidad de satisfacer nuevas funcionalidades o cambien las ya existentes, mientras que se procura no romper la compatibilidad con los clientes existentes y con el trabajo que hayan realizado.[11]

etcd es un almacén de valores clave altamente disponible utilizado para el almacenamiento permanente de todos los objetos REST API , como hemos explicado en el párrafo anterior que sucede. Los objetivos a alto nivel son principalmente el control de acceso y la fiabilidad de los datos. **API Server** solo proporciona acceso de lectura/escritura a **etcd**, ya que no queremos que el almacenamiento de API Server (etcd) quede expuesto al resto de nodos en nuestro

clúster o, lo que es peor, a Internet en general. Para que la fiabilidad de datos sea razonable, etcd debería ser ejecutado como un clúster, es decir, varias máquinas en las que cada una está ejecutando etcd formando un “almacén único”. Otra opción es ubicar todos estos estados en un almacenamiento duradero. Además, deberían de hacerse copias de seguridad periódicas del directorio de datos con la finalidad de que, si hay algún tipo de corrupción o fallo, reducir el tiempo de inactividad del sistema al mínimo posible y conseguir una recuperación rápida.[12]

El **administrador de control** (*Controller Manager*) [13] es un demonio *(daemon*) que se encarga de comprobar el correcto estado de un clúster a través del servidor de API y en caso de que no sea correcto el estado modificarlos que el objetivo de alcanzar el estado deseado.

El **planificador** (*scheduler)* [14] se encarga de determinar en qué nodo del sistema se va a ubicar un pod, por tanto es uno de los elementos más importantes en el sistema a la hora de conseguir un uso óptimo de los recursos. El planificador interno de Kubernetes (*kube-scheduler*) es altamente configurable, y dispone de dos tipos de reglas: predicados y funciones de prioridad.

Cada una de las reglas es aplicable a los nodos, los predicados son reglas que han de cumplirse para que se pueda seleccionar un nodo, por ello, los predicados son especialmente útiles cuando, por ejemplo, conocemos los requisitos mínimos de recursos que va a tener iniciar un pod y los recursos que quedan disponibles en cada nodo, de tal manera que con el predicado eliminamos los nodos que no fuese capaces de suplir las necesidades del pod. Este predicado es uno de los implementados por defecto (*PodFitsResources*). [15]

Las funciones de prioridad sirven para determinar de entre los nodos que han cumplido todos los predicados el nodo que queremos usar. Cada una de las funciones de prioridad da una puntuación desde 0 hasta 10 para un nodo y podemos combinar distintas funciones de prioridad dándole un peso a cada una. De entre las implementadas por Kubernetes caben destacar *BallancedResourceAllocation* y *LeastRequestedPriority*. [14] *BallancedResourceAllocation* intenta escoger un nodo que tenga una porcentaje de utilización de CPU tras desplegar un pod lo más parecido posible a la utilización de memoria. *LeastRequestedPriority* escoge el nodo que tras desplegar el pod dejaría la mayor fracción de recursos disponibles.

Si el planificador interno de Kubernetes no fuese capaz de suplir nuestras necesidades tenemos la posibilidad de hacer uno propio (o coger uno que ya esté hecho para Kubernetes por otra persona) e incluso utilizar varios planificadores diferentes a la vez. [16]

2.2 Nodo esclavo

Además del nodo de control explicado anteriormente, puede haber muchos más nodos con la finalidad de ejecutar el resto de los contenedores. Estos son también llamados minions o workers (trabajadores). No obstante, en cada nodo, existen varios elementos que son comunes a todos ellos. Estás elementos nos permiten comunicarnos con el nodo de control y administrar un nodo normal (**kubelet**), monitorizar los recursos que están siendo consumidos por un nodo en concreto (**cAdvisor**) y realizar conexiones proxy y balanceo de carga a nivel interno del sistema (**kube-proxy)**.

kubelet[17] es una agente que se encarga de gestionar un nodo. Existe un kubelet por nodo y este se encarga de gestionar todos los contenedores que han sido creados por Kubernetes. La configuración que reciben *(PodSpec)* puede darse como un archivo leído cada cierto tiempo, un extremo HTTP (*HyperText Transfer Protocol*) al que se accede cada cierto tiempo o actuar el kubelet como servidor HTTP que responde mediante una API. A través del API Server, cada kubelet actúa como enlace entre el nodo de control y un nodo normal.

cAdvisor[18] es una herramienta de monitorización que nos permite medir el uso de los recursos y rendimiento de los contenedores de un nodo que se encuentra incluida dentro del

kubelet. cAdvisor es capaz de detectar cuáles son los contenedores que se están ejecutando en un nodo y recopilar la información para poder obtener datos útiles. Además, existe la posibilidad de mostrar los datos recopilados visualmente accediendo a un puerto concreto, facilitando su comprensión y visión a nivel de nodo. Si queremos recopilar la información de todo el servidor tenemos disponibles pods como *Heapster* que nos permiten integrar la información que ha obtenido cada nodo mediante cAdvisor para obtener una visión general del clúster.

kube-proxy[19] es una mezcla de un servicio proxy y un balanceador de carga. Permite hacer seguimiento TCP (Transmission Control Protocol) /UDP(User Datagram Protocol) o incluso crear un DNS(Domain Name System) a nivel del clúster, permitiendo redirigir y repartir el trabajo que llega a un nodo. Hablaremos más adelante de ese balanceo de carga que nos permite realizar kube-proxy.

3. El acceso a Kubernetes

Una vez hemos comprendido a grandes rasgos cuál es la estructura detrás de Kubernetes, es obvio que el administrador del servidor querrá acceder a las funcionalidades ofrecidas por el nodo de control para poder configurar el sistema como desee. Para ello dispone de dos herramientas: **kubectl** y **una página web que actúa como panel de control.**

kubectl[20] es una interfaz para mandar comandos a Kubernetes a través de la consola. Para poder usarlo, kubectl ha de ser instalado (no viene instalado por defecto) en la máquina que ejecuta Kubernetes. A kubectl se le pasa una serie de parámetros de los cuáles el primero es el comando a ejecutar, el tipo de recurso nombre del recurso y algunas banderas opcionales.

Por tanto, mediante kubectl, podemos, entre otras cosas: crear, iniciar, parar y eliminar imágenes de contenedores, hacer dichos contenedores accesibles desde el exterior, modificar distintos parámetros de ejecución, obtener información del clúster y obtener listas de servicios, pods y otros recursos activos.

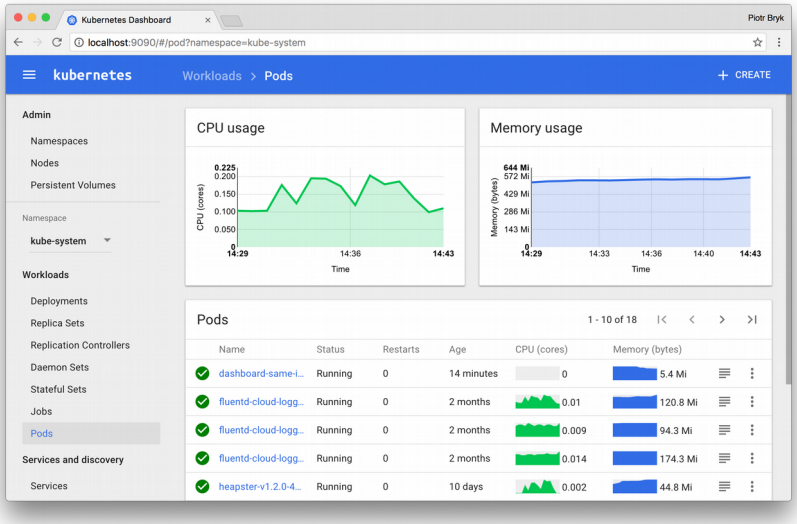

Figura 1: Visión de pods y monitorización de recursos en la interfaz web

Además de kubectl, existe la posibilidad de acceder a través de una interfaz web (tal y como podemos observar en la figura 1) que nos permite realizar operaciones similares (aunque no todas) de una manera más cómoda. Algunas de las cosas más importantes que puede hacer son lanzar un pod nuevo, ver que pods están siendo ejecutados, modificar algunos parámetros de configuración o consultar los nodos que están siendo usados.

No obstante, la característica más útil de la interfaz web es el hecho de que nos permita visualizar la monitorización de todo el sistema y de cada pod en concreto de una manera gráfica.

4. El autoescalado en Kubernetes

Una de las características más interesantes proporcionado por Kubernetes es el autoescalado, tanto a nivel de nodos como a nivel de pods. El autoescalado nos permite ajustar automáticamente el número de nodos y de pods que se encuentran activos en la plataforma Kubernetes.[21] Así pues cuando tenemos alguna hora en la que haya más carga de lo habitual o menos se crearán más nodos y pods y si recibimos mucha menos quitamos pods y/o nodos con el fin de no malgastar recursos.

Existen dos tipos de escalado: el escalado **horizontal**, que aumenta el número de instancias de un componente, y el escalado **vertical**, que aumenta los recursos que puede utilizar un componente. Vistos desde el punto de vista de Kubernetes los componentes que podemos escalar son los nodos y los pods. En este trabajo vamos a centrarnos en los elementos que nos permiten un escalado automático horizontal.

En Kubernetes, existen dos herramientas que nos permiten realizar un escalado horizontal:: **Cluster Autoscaler** y **Horizontal Pod Scaling**.

En clusters de GCE (*Google Compute Engine*) o GKE (Google Container Engine), que son dos plataformas de Google para alquilar servidores escalables (la primera utilizando máquinas virtuales y la segunda utilizando contenedores Docker) que utilizan Kubernetes, podemos utilizar el **autoescalado de cluster** (*Cluster Autoscaler).*[22] nos permite configurar si se producirá autoescalado basándose en la necesidades creadas por los pods. Si tuviésemos el clúster saturado y hay pods que no han podido ser planificados, es decir, no cumplían todos los predicados impuestos en alguno de los nodos (por ejemplo, no hay suficiente CPU disponible o memoria), comprueba si añadiendo un nuevo nodo al clúster podemos satisfacer las necesidades de los pods que no habíamos podido crear. Es importante recordar de que en caso de que no haya suficientes recursos disponibles el pod se queda a la espera para poder ejecutarse. Además de este autoescalado al alza, también realiza autoescalado a la baja, con el fin de liberar recursos que no se están usando, si un nodo no es necesitado durante 10 minutos o más el autoescalado de cluster eliminará el nodo. Además, podemos configurar los casos extremos indicando el número máximo y mínimo de nodos que vamos a necesitar.

La otra forma de autoescalado que vemos es a nivel de pod y es el **escalado horizontal de pods** (*Horizontal Pod Scaling*) [23] y se aplica al controlador de replicación, Deployments y ReplicaSet basándose en la utilización media de CPU por cada contenedor en el pod. En este caso en vez de añadir nuevos nodos se intenta añadir o quitar réplicas del pod con el objetivo de obtener un valor lo más cercano posible a la utilización media de CPU indicada. De la misma manera que en el caso anterior podemos indicar el número máximo de réplicas que permitimos que estén creadas y el número mínimo que debe de haber.

5. El balanceo de carga en Kubernetes

La otra característica interesante de Kubernetes en la que vamos a hacer hincapié es el balanceador de carga. El balanceo de carga [24] es la forma en la que un servidor utiliza un mecanismo o herramienta al que denominamos “balanceador de carga” con la finalidad de repartir las peticiones que llegan al mismo entre varios servidores a los que se encuentra conectado con el fin, habitualmente, de evitar que se sature el sistema y, por tanto, el servicio que el servidor estaba proporcionando quede temporalmente inaccesible.

Tal y como mencionamos brevemente cuando explicamos los componentes de un nodo normal, de forma nativa y a nivel interno Kubernetes proporciona **kube-proxy**[19] como una posible herramienta para el balanceo de carga. kube-proxy nos permite realizar balanceo de carga de servicios que utilizan protocolos TCP y UDP de una forma directa de un nodo a otro o utilizando un algoritmo *Round Robin* para determinar qué nodo ha de ser seleccionado para atender la carga. El algoritmo *Round Robin* implementado en kube-proxy simplemente alterna entre los nodos disponibles. Aunque parezca que sea excesivamente simple se asegura de que el nodo escogido se encuentre en un buen estado, es decir, no esté saturado y sea capaz de atender la petición y de que una vez a un cliente se le asigna un nodo, la dirección IP de dicho cliente queda asociada a ese nodo del servidor mientras dure la sesión. Por tanto, kube-proxy nos ofrece una manera simple de tener un balanceador de carga sin ninguna complicación nada más instalar Kubernetes en una máquina.

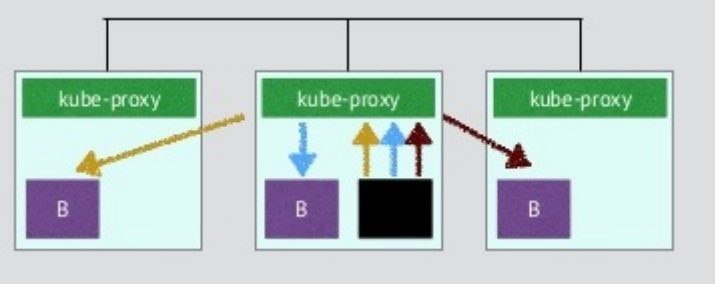

Figura 2: kube-proxy del nodo central redistribuye el trabajo a otras réplicas del pod B en otros nodos

No obstante, kube-proxy nos proporciona un servicio a nivel interno de Kubernetes, y a veces nos interesa exponer el balanceo de carga al exterior. Podemos hacerlo de varias maneras.

Mediante un servicio configurado como **nodePort** [25], el servicio estará accesible a través de un único puerto (también podemos especificar un rango de puertos y que Kubernetes escoja uno que esté disponible) pero será accesible desde cualquier nodo. Así pues una vez llegue la petición desde el exterior a dicho puerto, a través de kube-proxy se determinará de manera similar al caso anterior qué nodo ha de procesar la petición. Por tanto, la única diferencia que existe entre la opción interna con kube-proxy y nodePort es que podemos acceder al servicio desde cualquier nodo del clúster.

Otra opción que se basa en kube-proxy para realizar el balanceo de carga es mediante External IPs. Así pues cuando a un servicio se le asigna varias **External IPs** [26], va a existir un nodo por cada IP propuesta que va a permitir acceder a ese servicio. Esta opción nos permite trabajar de una manera similar a nodePort pero sólo en un número limitado de pods con acceso externo.

Otra opción de la que disponemos es la de un servicio configurado como **LoadBalancer** [27] que se encuentra en algún lugar en la nube externo. Así pues, un sistema externo se encarga de redistribuir el tráfico a cada uno de los pods que se encuentran dentro de los nodos de Kubernetes, y por tanto quitando el *overhead* (tiempo extra consumido) provocado por el balanceo de la carga.

La última opción de la que vamos a hablar es mediante **Ingress** [28], Mediante esta podemos configurar un balanceador de carga externo HTTP mediante varias URL (*Uniform Resource Location*) dentro de un dominio (ejemplo: http://mihost/servicio1 http://mihost/servicio2) o mediante varios nombres de dominio (ejemplo: http://servicio1.mihost http://servicio2.mihost). Esta solución es la más completa ofrecida por Kubernetes ya que además de las características mencionadas anteriormente obtenemos la posibilidad de añadir terminaciones SSL permitiendo ofertar servicios más seguros. No obstante, es necesario poner un controlador que se encargue de administrar los recursos de tipo Ingress, por tanto, podemos crear uno

nosotros si así lo vemos conveniente o usar alguno de tipo software como NGINX [29] que nos permiten realizar configuraciones mucho más avanzadas.

6. Kubernetes en la actualidad

En la actualidad, las empresas pueden beneficiarse en gran medida de esta plataforma desarrollada por Google. No tienen que preocuparse ni de la ejecución de los contenedores (Docker u otros) ni de la propia orquestación y localización de los mismos (Kubernetes). Solamente de que sus aplicaciones se ejecutan como ellos esperan. Estas empresas conseguirán ventajas con este sistema de contenedores en la nube tales como mayor eficiencia, flexibilidad, rapidez, facilidad para el cuidado y mantenimiento y, lo que es más importante, escalabilidad.[30]

Por si todo esto fuera poco, y como ya sabemos, se trata de una plataforma con código abierto. Entonces, habrá empresas que lo utilicen directamente para sus proyectos y que se animen a establecerlo en su propio sistema proporcionando su propio soporte. Algunas de esas empresas que se han empezado a usar y dar soporte para Kubernetes son *Red Hat*, *IBM*, *CoreOS* y últimamente *Microsoft*.

Microsoft Azure es una colección cada vez mayor de servicios integrados en la nube. Actualmente, incorpora la ejecución de contenedores con integración de Docker [31]. Esto quiere decir que existe un apoyo para Kubernetes en la infraestructura de Azure, en el cual está basado Azure Container Service (ACS). Kubernetes 1.4 ya ofrece soporte para redes Azure nativas, balanceo de carga e integración de discos Azure.[32]

En Red Hat tenemos *Red Hat Enterprise Linux Atomic Host*, es un administrador de contenedores que utiliza también Docker y Kubernetes[33]. Sin embargo, *Red Hat Enterprise Linux Atomic Host* trata de desarrollar un sistema operativo minimizado que está optimizado exclusivamente para dicha ejecución y administración de contenedores[34].

CoreOS es un sistema operativo ligero de código abierto basado en el kernel de linux, utiliza Kubernetes para administrar la infraestructura de sus contenedores[35]. Es decir, utiliza Kubernetes, no para un proyecto concreto, sino para desarrollar su propio sistema.

Como se ha mencionado antes, también tenemos empresas que utilizan directamente Kubernetes para realizar proyectos concretos. Algunos ejemplos son *Pearson*, *eBay*, *Wikimedia* o *Box* entre muchos otros. *Pearson* es una compañía educativa que atiende a 75 millones de estudiantes en todo el mundo. Se propuso subir esa cifra a 200 millones para 2025. Para conseguir este objetivo, le era fundamental apoyarse en una plataforma con una infraestructura que le permitiera ganar flexibilidad y, sobretodo, escalabilidad. Utilizan Kubernetes para facilitar la adaptación y trabajo de los desarrolladores y mejorar su productividad.[36]

En la documentación oficial de Kubernetes[37] podemos encontrar una referencia que nos aporta información sobre *eBay*[38]. Es una empresa especializada en subastas online con 159 millones de compradores activos en 190 países y más de 800 millones de productos registrados en su servicio. La escalabilidad en un sistema que cada día crece más es muy importante. Ese es el primer motivo por el que se beneficia de las ventajas que proporciona Kubernetes. Al principio utilizaba OpenStack[39]; un software que, en resumen, sirve para controlar grandes grupos de computación, almacenamiento y red a través de un centro de datos. En la actualidad, *eBay* está intercalando Kubernetes con sus nubes de OpenStack para administrar las aplicaciones mediante contenedores.

Wikimedia[40] es una fundación sin propósito de lucro que pretende desarrollar un proyecto de edición de referencias en colaboración con el mundo, lo que incluye a la famosa *Wikipedia*. Para ayudar a los usuarios a mantener y utilizar estas wikis y crear un entorno de alojamiento para desarrolladores comunitarios. Para ello, se sirve de la ayuda que supone Kubernetes porque puede imitar flujos de trabajos existentes (sistema de revisiones), al mismo tiempo que

reduce la complejidad del sistema previamente implementado para *Wikimedia Tool Labs*. Estos desarrolladores o usuarios voluntarios podrán utilizar herramientas y bots desarrollados a partir de esta plataforma de una forma más sencilla y ejecutar wikis en todo el mundo, sin mencionar que ayuda a reducir el vandalismo por parte de usuarios que proporcionan información incorrecta o falsa. Al principio, resultaba mucho más caótico realizar estas herramientas y bots para estas personas.

Por último, tenemos en la documentación de Kubernetes de nuevo[37] una referencia, esta vez a un blog sobre *Box* [41]. Cada vez que querían desarrollar un solo servicio le llevaba a los equipos de la empresa una cantidad de tiempo indecente y, por si fuera poco, debían de enfrentarse y solucionar las incoherencias que surgían durante su desarrollo. Por ello, esta empresa decidió apoyarse también en Kubernetes, ya que disponían de unas herramientas muy valiosas para realizar este trabajo, construyendo su plataforma alrededor de esta tecnología de contenedores. Gracias a esto le resulta más fácil a los desarrolladores llevar su servicio y desplegarlo en entornos de desarrollo, puesta en escena y producción.

En la actualidad no solo existe **Kubernetes** como único orquestador de contenedores. Tenemos otros que tienen el mismo fin, aunque con algunas diferencias entre ellos. De entre ellos destaca **Docker Swarm**.

Además de todos las ventajas que tiene **Kubernetes**, hay ciertos aspectos que no son tan positivos. Tenemos que tener en cuenta que utiliza un CLI *(Command Line Interface)* distinto al del propio Docker y diferentes API *(Application Programming Interface)*. En otras palabras, a pesar de ser un nivel de abstracción por encima de Docker, no puede usar a éste para definir contenedores, tiene que diseñarse todo desde cero exclusivamente para Kubernetes.

Por otra parte, **Docker Swarm** [42] se enfocó de forma distinta, ya que proviene de la propia compañía de Docker y por tanto se centró en un un clústering nativo para el mismo. Utiliza el CLI nativo de Docker por lo que las herramientas que se utilizaban para comunicarse con Docker (Docker CLI, Docker Compose, Dokku, Krane, etc.) siguen funcionando para este orquestador y no se tiene que diseñar o aprender todo desde el principio. Esto es una ventaja y una desventaja al mismo tiempo ya que el propio Docker no es perfecto y estamos expuestos a sus limitaciones. Si la API de Docker tiene algún problema o no admite hacer algo, tendremos ese mismo problema con Docker Swarm.

7. Conclusión

Como hemos podido observar, la plataforma de orquestación de contenedores creada por Google, está teniendo un gran impacto en la actualidad. El hecho de que el proyecto esté respaldado por una compañía de esa categoría, la facilidad puesta a la hora de modificar el código si es necesario y la gran cantidad de abstracciones que han sido puestas para facilitar la administración del clúster por no hablar de que durante 15 años ha estado funcionando como plataforma de orquestación de contenedores para Google de forma privada.

Aunque nos hemos centrado en dos de las características que nos han parecido más interesantes a la hora de estudiar Kubernetes (el autoescalado y el balanceo de carga), el proyecto es mucho más amplio y existen otras muchas opciones interesantes a escoger de las que podíamos haber hablado en profundidad (como el sistema reacciona ante fallos, la configuración de un pod, el sistema de revisiones sobre modificaciones, la ejecución de trabajos batch, la depuración de contenedores, etc.) pero no hemos podido porque extenderían excesivamente el trabajo.

Para acabar, creemos que, como hemos podido observar a la hora de estudiar cuál ha sido su impacto en la actualidad el hecho de que cada vez más empresas de grandes proporciones, su facilidad de instalación y uso en cualquier máquina y el hecho de que servicios de alquiler de

servidores virtualizados como Google Cloud Engine o Google Compute Engine van a permitir a Kubernetes seguir creciendo en el mundo de la ingeniería de servidores.

Referencias:

[1] http://kubernetes.io/docs/whatisk8s/ consultado el 6 de diciembre de 2016
[2] https://linuxcontainers.org/ consultado el 6 de diciembre de 2016
[3] http://kubernetes.io/docs/user-guide/pods/ consultado el 6 de diciembre de 2016
[4] http://kubernetes.io/docs/user-guide/labels/ consultado el 6 de diciembre de 2016
[5] http://kubernetes.io/docs/user-guide/replication-controller/ consultado el 6 de diciembre de 2016
[6] http://kubernetes.io/docs/user-guide/replicasets/ consultado el 14 de Diciembre de 2016
[7] http://kubernetes.io/docs/user-guide/deployments/ consultado el 14 de Diciembre de 2016
[8] http://kubernetes.io/docs/user-guide/services/ consultado el 6 de Diciembre de 2016
[9] https://github.com/kubernetes/kubernetes/blob/master/docs/design/architecture.md consultado el 8 de Diciembre de 2016
[10] http://kubernetes.io/docs/admin/kube-apiserver/ 24 de Octubre de 2016
[11] http://kubernetes.io/docs/api/ consultado el 7 de Diciembre de 2016
[12] http://kubernetes.io/docs/admin/etcd/ consultado el 7 de Diciembre de 2016
[13] http://kubernetes.io/docs/admin/kube-controller-manager/ consultado el 7 de Diciembre de 2016
[14] https://github.com/kubernetes/kubernetes/blob/master/docs/devel/scheduler.md consultado el 7 de Diciembre de 2016
[15] https://github.com/kubernetes/kubernetes/blob/master/docs/devel/scheduler_algorithm.md consultado el 8 de Diciembre de 2016
[16] http://kubernetes.io/docs/admin/multiple-schedulers/ consultado el 7 de Diciembre de 2016
[17] http://kubernetes.io/docs/admin/kubelet/ consultado el 13 de Diciembre de 2016
[18] http://blog.kubernetes.io/2015/05/resource-usage-monitoring-kubernetes.html consultado el 13 de Diciembre de 2016
[19] http://kubernetes.io/docs/admin/kube-proxy/ consultado el 13 de Diciembre de 2016
[20] http://kubernetes.io/docs/user-guide/kubectl-overview/ consultado el 13 de Diciembre de 2016
[21] http://blog.kubernetes.io/2016/07/autoscaling-in-kubernetes.html consultado el 13 de Diciembre de 2016
[22] http://kubernetes.io/docs/admin/cluster-management/#cluster-autoscaling consultado el 13 de Diciembre de 2016
[23] http://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/ consultado el 13 de Diciembre de 2016
[24]https://www.1and1.es/digitalguide/servidores/know-how/balanceo-de-carga-conoce-a-fondo-sus-ventajas/ consultado el 14 de Diciembre de 2016
[25] http://kubernetes.io/docs/user-guide/services/#type-nodeport consultado el 15 de Diciembre de 2016

[26] http://kubernetes.io/docs/user-guide/services/#external-ips consultado el 15 de Diciembre de 2016
[27] http://kubernetes.io/docs/user-guide/services/#type-loadbalancer consultado el 15 de Diciembre de 2016
[28] http://kubernetes.io/docs/user-guide/ingress/ consultado el 16 de Diciembre de 2016
[29] https://github.com/nginxinc/kubernetes-ingress consultado el 16 de Diciembre de 2016
[30]https://blog.1and1.es/2016/08/19/kubernetes-orquestacion-de-contenedores-para-el-futuro/, 19 de Agosto de 2016
[31]https://azure.microsoft.com/es-es/overview/what-is-azure/ Consultado el 15 de Diciembre de 2016
[32]https://azure.microsoft.com/en-us/blog/azure-container-service-the-cloud-s-most-open-option-for-containers/ 7 de Noviembre de 2016.
[33]https://www.redhat.com/en/services/training/rh270-managing-containers-red-hat-enterprise-linux-atomic-host Consultado el 15 de Diciembre de 2016.
[34]https://access.redhat.com/products/red-hat-enterprise-linux/atomic-host-beta Consultado el 15 de Diciembre de 2016
[35]https://coreos.com/kubernetes/docs/latest/ Consultado el 15 de Diciembre de 2016
[36]http://kubernetes.io/case-studies/pearson/ Consultado el 16 de Diciembre de 2016
[37]http://kubernetes.io/case-studies/ Consultado el 16 de Diciembre de 2016
[38]https://www.nextplatform.com/2015/11/12/inside-ebays-shift-to-kubernetes-and-containers-atop-openstack/ Consultado el 16 de Diciembre de 2016
[39]https://www.openstack.org/es_ES/ Consultado el 16 de Diciembre de 2016
[40]http://kubernetes.io/case-studies/wikimedia/ Consultado el 16 de Diciembre de 2016
[41]https://blog.box.com/blog/kubernetes-box-microservices-maximum-velocity/ 22 de Julio de 2016
[42]https://docs.docker.com/swarm/overview/ Consultado el 17 de Diciembre de 2016