Universidad Zaragoza

Universidad Zaragoza

EVALUACIÓN DE LA DEGRADACIÓN DE PRESTACIONES EN ESCENARIOS MULTI-APLICACIÓN EN KUBERNETES

Carlos Javier Tolón Martín

Director: Víctor Medel Gracia

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Junio de 2017

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. Carlos Javier Tolón Martín,

con nº de DNI 72999951K en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) Grado en Ingeniería Informática, (Título del Trabajo) EVALUACIÓN DE LA DEGRADACIÓN DE PRESTACIONES EN ESCENARIOS MULTI-APLICACIÓN EN KUBERNETES.

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 21 de Junio de 2017

Fdo: Carlos Javier Tolón Martín

EVALUACIÓN DE LA DEGRADACIÓN DE PRESTACIONES EN ESCENARIOS MULTI-APLICACIÓN EN KUBERNETES

RESUMEN

Este proyecto se desarrolla en el ámbito de las aplicaciones distribuidas en sistemas de contenedores. Los objetivos parten de la hipótesis de que la compartición de recursos por distintos contenedores provoca la degradación de prestaciones. Para demostrar esta hipótesis se han realizado dos conjuntos de experimentos. El primero con aplicaciones que realizan un uso intensivo de un recurso, como CPU o disco. En cambio, en el segundo se utilizan aplicaciones industriales de procesado de *data streaming* de *MapReduce* y *Web graph* (WordCount y PageRank respectivamente). Ambos conjuntos de experimentos han puesto de manifiesto la existencia de degradación de prestaciones en las diferentes configuraciones. Esta degradación puede llegar a ser del 300 %.

Probada la existencia de la degradación, se ha planteado e implementado una solución que consiste en un *scheduler* de cliente. El usuario realiza una caracterización de su aplicación indicando el recurso que más requiere y su nivel de uso. De esta forma, el *scheduler* decide el nodo destino. Para decidir la máquina sobre la que se despliega, se balancea al máximo el número de aplicaciones por nodo y se minimiza la penalización de unir varias aplicaciones que realicen uso de un mismo recurso. Con esta solución se acerca el proceso de *scheduling* de aplicaciones al usuario y se obtiene una mejora de rendimiento.

Se ha probado que los mecanismos de aislamiento de recursos que proporciona Kubernetes (etiquetas *limits* y *request*) resultan insuficientes para mejorar las prestaciones de las aplicaciones. El *scheduler* propuesto como solución ha sido validado y comparado con el *scheduler* por defecto de Kubernetes mostrando una mejoría del 20 % en el caso medio del *scheduler* por defecto y llegando a mejorar en un 32 % el peor caso del mismo.

Finalmente, se ha documentado el proceso de instalación de Kubernetes y se ha explicado la operativa y puesta en marcha de dos *frameworks* de procesado de *data streaming* sobre Kubernetes, como son Flink y Thrill, que han sido utilizados para la demostración de la hipótesis planteada.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El paradigma de computación distribuida de *cloud* permite que la gestión de recursos compartidos y sus implicaciones sean transparentes tanto para los usuarios como para los desarrolladores de aplicaciones. El escenario más común en este paradigma supone que múltiples usuarios ejecuten diferentes aplicaciones sobre un conjunto de recursos, tradicionalmente máquinas virtuales (MV), cada cual con diferentes requerimientos económicos y de calidad de servicio. Sistemas como *Amazon Web Services* o *Google Cloud Platform* permiten abstraer todos estos aspectos, a la vez que evitan la necesidad de realizar un gasto de compra de infraestructura.

Durante años, el referente industrial en los Sistemas de Infraestructura como Servicio (IaaS por sus siglas en inglés) [1] han sido las máquinas virtuales; sin embargo, en los últimos años ha aparecido una tecnología que ha comenzado a competir con ellas, los contenedores. La abstracción de contenedor empaqueta todos los requisitos para la ejecución de una aplicación. En comparación con las máquinas virtuales, los contenedores no envuelven un SO completo sino que almacenan las librerías y configuraciones necesarias para que un *software* funcione correctamente.

La tecnología de contenedores proporciona una forma ligera y flexible [2] de desplegar aplicaciones y permite alcanzar ratios de uso de recursos más altos que las máquinas virtuales. Esta flexibilidad produce un menor aislamiento de los recursos del sistema [3] y, también, a nivel de seguridad [4], ya que no hay separación entre el *kernel* del *host* y el propio contenedor. Esta flexibilidad y falta de aislamiento entre los contenedores desplegados, se traduce en una compartición de recursos de la máquina. Esta compartición, produce una degradación en las prestaciones de las aplicaciones.

Por otra parte, en los últimos años la disciplina dedicada al procesado de datos masivos ha sufrido un gran crecimiento dentro del sector de las Tecnologías de la Información y la Comunicación. Esta disciplina, conocida como *Big Data*, requiere de un almacena-

miento optimizado de los datos y de un procesamiento, con el objetivo de crear informes estadísticos y modelos predictivos dentro de dicho sector. Determinadas aplicaciones dentro del *Big Data* requieren de un procesado en el menor tiempo posible, con el fin de proporcionar información al usuario lo antes posible. Este paradigma de computación se conoce como procesado de *data streaming*. Las altas necesidades computacionales y de escalado, unidas con la flexibilidad del *cloud*, hace que las aplicaciones de procesado de *data streaming* sean un caso de uso habitual del *cloud*.

El presente Trabajo Fin de Grado consta de seis capítulos. En el Capítulo 1 se introduce el problema analizado y se explica la motivación y objetivos del proyecto. En el Capítulo 2 se plantea el *background* tecnológico y una aproximación al estado del arte; en el Capítulo 3, por su parte, se describe la arquitectura del sistema a contemplar. Tras plantear el sistema, se explica, en el Capítulo 4, el diseño experimental que tiene como objetivo demostrar empíricamente la degradación de prestaciones. En base a dichas evidencias, en el Capítulo 5, se presenta una posible solución para la mejora del rendimiento. Finalmente, en el Capítulo 6, se presentan las conclusiones de este trabajo.

Como complemento, se incluyen cuatro anexos de carácter técnico. El Anexo A explica el despliegue realizado así como los comandos más importantes a la hora del manejo del *cluster*; en el Anexo B se presenta el *framework* Flink y se explica el despliegue que se ha realizado sobre Kubernetes; en el Anexo C, de la misma forma, se presenta Thrill y el código para su despliegue sobre Kubernetes; finalmente, en el Anexo D se muestra el código fuente del *scheduler* de cliente que presentado como solución.

1.1. Motivación

En el contexto actual, los sistemas de contenedores están cobrando cada vez más importancia. La relativa novedad de la tecnología hace que surjan una serie de problemas y retos susceptibles de ser estudiados. Uno de estos retos es determinar cómo varios contenedores comparten recursos en una misma máquina física y cómo afecta esto a las aplicaciones. Las implicaciones económicas de cualquier optimización a las prestaciones resultan inmediatas. Por otro lado, las soluciones propuestas deben seguir manteniendo la transparencia exigida en el paradigma *cloud*.

Desde el ámbito de la investigación se han propuesto mecanismos que incentivan la baja interferencia de ciertas aplicaciones, premiando aquellas que hiciesen un mejor y correcto uso de los recursos y penalizando a las que consumiesen casi todos los recursos de forma continuada. Esta es la línea que se definen en los trabajos de Parangon [5] y ARQ [6]. Otra propuesta, defendida en el presente trabajo, consiste en que el usuario caracterice de manera informal sus aplicaciones para guiar al proveedor a tomar decisiones más eficientes.

Dentro del presente trabajo se han elegido las aplicaciones de procesado de *data streaming* debido a que tienen unos requerimientos de tiempo muy altos y una degradación de prestaciones en las mismas provocaría que los resultados de sus análisis ya no sean válidos en caso de llegar demasiado tarde. Por tanto, ¿cómo reaccionarán en caso de compartir una máquina física con otros usuarios que están ejecutando sus aplicaciones?

1.2. Objetivos

En el contexto de la elaboración del presente Trabajo fin de Grado, se definen los siguientes objetivos:

- Demostración empírica de que la ejecución de diferentes contenedores sobre una misma máquina física provoca una degradación en las prestaciones.
- Diseño de un conjunto de experimentos basado en aplicaciones reales para cuantificar la degradación.
- Diseño de una arquitectura basada en contenedores que permita técnicas de *scheduling* por parte del cliente. De esta manera, a través de una caracterización previa de aplicaciones se pretende reducir la degradación de prestaciones.
- Validación experimental de la propuesta de *scheduling* de cliente y comparación con el *scheduler* por defecto de Kubernetes.
- Análisis de las implicaciones de la degradación en escenarios reales del ámbito del procesado de *data streaming*.
- Aprendizaje, conocimiento y documentación de la puesta en marcha y de las implicaciones de un sistema de gestión de contenedores (Kubernetes) y de diversos *frameworks* de procesamiento de *data streaming*.
- Conocimiento y puesta en práctica de una metodología eminentemente científica basada en el planteamiento de hipótesis y en su validación experimental.

Capítulo 2

Background tecnológico y estado del arte

A lo largo del presente trabajo se manejan una serie de conceptos tecnológicos que tienen una fuerte implicación en el objeto central del estudio. Por tanto, resulta fundamental presentar una serie de elementos previos a fin de entender por qué se va a producir la degradación de prestaciones. Primero se introducirán las nuevas tecnologías de contenedores, señalando las principales diferencias con las máquinas virtuales. Después, se presentarán diferentes *frameworks* de procesamiento de *data streaming* para los que los requerimientos de calidad de servicio son muy importantes. Finalmente, en la Sección 2.3, se presentarán algunos trabajos relacionados con al gestión de recursos.

2.1. Sistemas de contenedores

La organización Linux Containers (LXC) [7] proporciona todas las herramientas necesarias para el desarrollo de contenedores. El principal conjunto de herramientas, plantillas y librerías es LXC que tiene como objetivo la creación de un entorno lo más cercano posible a un Linux estándar pero sin un *kernel* separado.

Existen varias plataformas para el desarrollo de contenedores, aunque la que está más extendida en el mercado es Docker [8]. Docker utiliza LXC [9] para encapsular aplicaciones. Otras alternativas como OpenVZ [10], FreeBSD jails [11], Solaris Zone [12] o Rocket [13] no han sido capaces de alcanzar a Docker.

Existe un contraste entre MVs y contenedores, ya que las MVs son un SO aislado, sobre el que es sencillo aplicar políticas de red, de usuarios, de seguridad, etc. Este tipo de virtualización proporciona una reducción de costes o simplificación en el mantenimiento

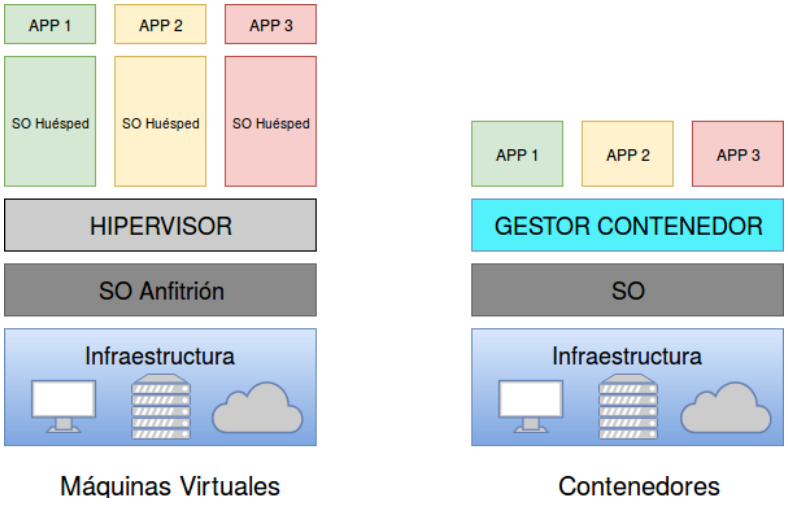


Figura 2.1: Comparación entre Máquinas Virtuales y Contenedores.

y ofrece la posibilidad de compartir una misma máquina física con varias virtuales [14].

En cambio los contenedores pueden implementarse y desplegarse sin un SO huésped, en un corto periodo de tiempo y con un uso de recursos económico. Además, requieren menos espacio de almacenamiento ya que solo contienen recursos *software* y no un SO completo. Otra métrica a favor de los contenedores es el tiempo de arranque, que puede llegar a ser 10 veces menor que en un máquina virtual [15]. En la Figura 2.1 se puede observar como los contenedores empaquetan todo lo necesario para la ejecución de una aplicación mientras que las MVs son un SO completo.

Los sistemas de contenedores utilizan, principalmente, dos mecanismos [16]. El primero son los *Linux Groups*, que permiten agregar/separar un conjunto de tareas, y a todos sus hijos, en una jerarquía de grupos con un comportamiento especial. El segundo mecanismo, los *Namespaces*, permiten aislar diferentes áreas como la red o el espacio de proceso. Es decir, aisla ciertos recursos y solo los procesos que pertenezcan a ese *namespace* pueden utilizarlos.

Los sistemas de contenedores son más flexibles que las MVs debido a la limitación de recursos en la definición de las mismas. Esta limitación no es necesaria en los contenedores, que almacenan todo lo necesario para la ejecución de una aplicación. Todas estas plataformas permiten el manejo y despliegue de aplicaciones en contenedores aislados pero, en cambio, no proporcionan posibilidades de más alto nivel como escalado, *scheduling* o gestión de un *cluster*.

En un nivel de abstracción superior se encuentran los sistemas de gestión de contenedores. Una de las opciones más utilizadas a día de hoy es Kubernetes[1], un sistema de

[1] `https://kubernetes.io/`

código abierto para el despliegue automático de aplicaciones en contenedores, que incluye un nivel más de abstracción denominado *pod*, con sencillez para el escalado y manejo de un *cluster*. Este sistema, se compone de un maestro que elige, a través de su *scheduler*, sobre qué máquina esclavo se va a desplegar cada contenedor (en el Anexo A se explica el funcionamiento básico de Kubernetes y en la Sección 3.2 se explica su *scheduler* por defecto).

Otro sistema para el manejo de un *cluster* con contenedores es *Docker Swarm* [17], el cual convierte un conjunto de máquinas Docker en un único *host* virtual. Debido a que utiliza la API estándar de Docker, cualquier petición que se hace sobre un demonio de Docker puede hacerse sobre Swarm comunicándose de forma transparente al *cluster* de máquinas.

Debido a la flexibilidad de estos sistemas, los recursos de la máquina sobre las que se despliegan los contenedores son compartidos por todos ellos, por lo que van a competir por estos mientras ejecutan la aplicaciones para la que estén destinados.

Además, existe otro sistema en un nivel de abstracción superior, OpenShift [18]. Es una plataforma de aplicaciones que integra de forma nativa Docker y Kubernetes. Proporciona suministro, gestión y escalado de aplicaciones en contenedores.

2.2. Sistemas de procesado de *data streaming*

En un mundo globalizado en el que los datos llevan años siendo un negocio muy importante, el procesado de *Big Data* [19] es clave para la mayoría de empresas e incluso para particulares. Es por esto, que las aplicaciones que se van a estudiar son las de *data streaming* [20], que analizan gran cantidad de datos en flujo. Reciben datos de forma simultánea y en pequeños tamaños, que deben ser procesados en el menor tiempo posible y de forma incremental para finalmente obtener una serie de resultados a analizar.

A nivel tecnológico e industrial, el interés por el procesado en *data streaming* se ha traducido en la aparición de un gran número de sistemas o *frameworks* para su procesado. Una de las más conocidas es **Spark** [21], aunque en su origen fue diseñado para procesar lotes de trabajos de manera más eficiente que Hadoop. Es por ello que se plantean otros *frameworks* que si que son fueron diseñados en origen para el procesado de *data streaming*. El primero es **Flink** [22], un *framework* desarrollado por Apache en JAVA, para aplicaciones de procesado de *data streaming*. La operativa de Flink se explica en el Anexo B. Otro *framework* más reciente es **Thrill**, similar a Flink pero escrito en C++. Este sistema tiene un funcionamiento diferente y se explica en el Anexo C. Finalmente, **Storm** [23] es un sistema de procesado de *data streaming* en tiempo real. Es un sistema escalable de forma sencilla ya que su uso se centra en aplicaciones que necesiten nuevos nodos para el procesado de más datos, por ejemplo Twitter.

En contraste con el procesado de *Data streaming*, está el procesado en lotes o *batch*, en el que Hadoop se ha convertido en un referente. Se analizan una serie de datos finitos y no suele existir un requerimiento tan alto en tiempo como el procesado en *streaming*. Ambos tipos de procesado, toman como base la Arquitectura Lambda [24], que define una serie de principios para los sistemas de procesado de datos.

2.3. Estado del arte

El provisionamiento de MVs en el contexto de los IaaS propone una forma de virtualizar tanto el *hardware* como el *software* de un sistema. Existen una serie de debilidades en estos sistemas; en especial que puede haber recursos que se desaprovechan. Por otra parte, sistemas con requisitos de rendimiento muy altos precisan de máquinas anfitrión muy potentes y, por tanto, muy caras, en las que puede que no se utilice el 100 % de los recursos todo el tiempo [15].

Desde el ámbito de las MVs, para tratar de paliar el desaprovechamiento de recursos, se han propuesto técnicas de *overbooking* complementadas con técnicas de migración para situaciones de uso intensivo de recursos [25]. En [26], se analiza el impacto en la degradación de prestaciones que tiene el *overbooking* de recursos físicos. Por otra parte, la interferencia que se produce entre máquinas virtuales que residen en un mismo *host* físico ha sido estudiada en [27]. En este trabajo se atienden principalmente a aquellos recursos dificilmente aislables y modelizables, como es la jerarquía de memoria. Además, analizan las implicaciones que dicho problema tiene en el escalado.

Partiendo de la experiencia de las MVs, las tecnologías de contenedores han sido estudiadas en comparación con las primeras. De esta manera, los trabajos que inciden en la comparación de rendimiento de ambos tipos de tecnologías son bastante frecuentes [28], [29], [30], [31]. Dentro del conocimiento del autor, la degradación de prestaciones entre diferentes contenedores desplegados en la máquina física no ha sido estudiada por la literatura. Debido a esto, resulta fundamental caracterizar dicha degradación de manera experimental.

Dentro del entorno del *cloud*, el estudio del *scheduling* de contenedores está en auge. Por ejemplo, Google ha desarrollado varios *schedulers* para grandes infraestructuras basándose en una arquitectura centralizada [32, 33]. Algunos trabajos han propuesto mejorar los algoritmos estándares en infraestructuras *cloud* como Kubernetes o Docker Swarm. Sin embargo, en [34], los autores indicar la falta de estudio sobre el manejo de recursos en contenedores y proponen un *framework* de *scheduling* que pueda usarse para aplicar políticas a medida de *scheduling*, principalmente, en entornos locales. En [35] los autores proponen un *scheduler* que mapee los contenedores con distintas generaciones de servidores, según los requisitos y propiedades aprendidos ejecutando los contenedores.

Capítulo 3

Arquitectura del sistema

Se considera un sistema en el que una serie de usuarios despliegan sus aplicaciones en contenedores sobre un *cluster* de Kubernetes. Los diferentes usuarios no tienen ningún control sobre que máquina física o virtual se despliegan sus aplicaciones; o dicho de otro modo, la gestión de recursos y el uso de los recursos de manera compartida resulta transparente para ellos. Cada uno de ellos puede tener unos requerimientos de prestaciones (o equivalentemente Calidad de Servicio) diferentes, aunque queda fuera del objeto del presente trabajo analizarlos. Desde el punto de vista de gestión del *cluster*, las aplicaciones que los usuarios despliegan en contenedores se tratan como cajas negras, desconociendo como se comportan. La arquitectura del sistema se presenta en la Figura 3.1.

Como los usuarios no tienen ningún control ni conocimiento sobre que máquina física o virtual están desplegadas sus aplicaciones, no conocen si su aplicación interfiere a otras o si los contenedores de otros usuarios provocan una degradación en su aplicación.

3.1. *Cluster* de Kubernetes

El *cluster* de Kubernetes está compuesto por el *Control plane*, que son el nodo maestro y los nodos trabajadores, sobre los que se despliegan las aplicaciones. El *Control plane* puede desplegarse sobre el nodo maestro o repartirse sobre más nodos. Está compuesto por el *API server*, una interfaz RESTful que sirve como punto de entrada al cluster; el *scheduler* para la asignación de los *pods* a los nodos; etcd, el sistema de almacenamiento del *cluster* y el *Controller manager*, que coordina y combina diferentes controladores como el *replication controller*.

Todos los nodos que componen el *cluster* poseen el servicio **kubelet**, que realiza la comunicación con el maestro y con los contenedores desplegados en dicho nodo. Otro

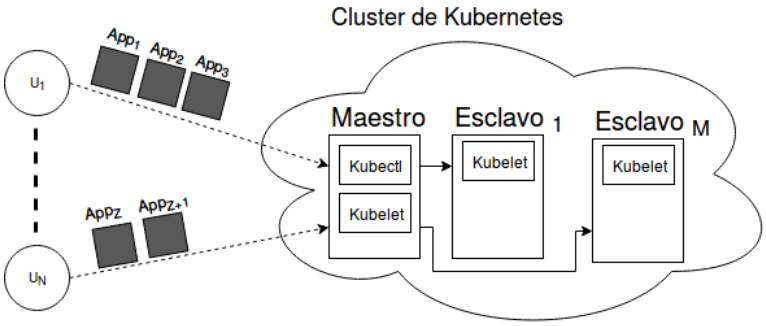


Figura 3.1: Representación Arquitectura del sistema.

componente básico es **kubectl**, que es una interfaz encargada de la comunicación entre el usuario y el cluster. Además, el *kube-proxy* se encarga de balancear las peticiones del usuario sobre todos los contenedores que configuran un servicio. La arquitectura de Kubernetes se presenta en la Figura 3.2 y en el Anexo A se explica la instalación y los comandos básicos de Kubernetes.

La unidad de despliegue de Kubernetes es el *pod*, una abstracción de un conjunto de contenedores con los mismos recursos compartidos (la interfaz de red y el sistema de almacenamiento). Cada *pod*, con todos sus contenedores, se ejecuta en un nodo, y tiene un dirección IP local compartida por todos los contenedores.

3.2. *Scheduler* de Kubernetes

El *scheduler* de Kubernetes [36] es una función que tiene un impacto significativo en la disponibilidad, rendimiento y capacidad. El *scheduler* tiene en cuenta tanto los recursos individuales como los colectivos, requerimientos en la calidad de servicio, restricciones *software* / *hardware* o políticas, etc.

El *scheduler* es un proceso desplegado sobre el maestro con otros componentes como el *API server*. Para cada *pod* debe encontrar un nodo en el que desplegarlo.

1. El primer paso consiste en aplicar una serie de "predicados" para filtrar los nodos que no son apropiados, por ejemplo si el *pod* tiene unos requisitos de recursos, el *scheduler* filtrará y eliminará los nodos que no cumplan esos requisitos.

2. En el segundo paso se asignan prioridades para clasificar los nodos que no fueron eliminados en el primer paso. Por lo tanto, se priorizan los nodos, teóricamente, favoreciendo aquellos con menor carga de trabajo.

3. Finalmente, se elige el nodo de menor prioridad, si hay varios con una misma prioridad se elige uno de ellos.

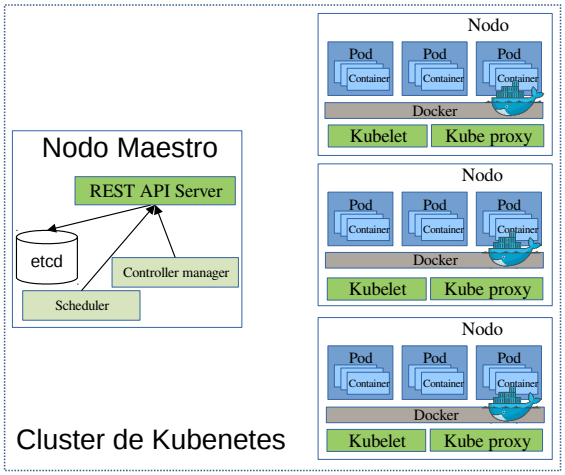


Figura 3.2: Arquitectura de Kubernetes.

Existen dos etiquetas que se pueden definir en el manifiesto de un *pod*. Los *limits*, indican el máximo que puede utilizar ese contenedor, mientras que *requests* son los recursos que se piden. Si los *requests* <*limits* se garantizan los pedidos (*request*), pero se puede llegar a los *limits* en caso de que otros contenedores no estén utilizando dichos recursos. La definición de los recursos es compleja ya que para definir un uso del 10 % de CPU se debe indicar con 100m (que son 100 "milicpu").

En todo este proceso, el *scheduler* no conoce el contenido del *pod*, simplemente comprueba si hay alguna restricción de recursos en el manifiesto de despliegue. Estos *pods* (de los cuales no se definen todos los recursos que piden) se conocen como **Best-Effort**. Pero existen dos tipos más, los **Guaranteed** que han sido definidos con todos *limits* y los *requests* iguales y los *pods* **Burstable** en caso de que algún *request* sea menor a los *limits*, o no se defina.

El *scheduler* de Kubernetes tiene un buen funcionamiento para los *pods* con los recursos definidos como *Guaranteed* o *Burstable*, pero la mayoría de *pods* definidos son *Best-Effort* lo que implica más complejidad.

3.3. Recursos compartidos en el sistema

Tras el proceso de selección de nodo para cada *pod*, sobre una misma máquina física se han podido desplegar numerosos contenedores que van a compartir todos sus recursos. Esta compartición supone una potencial degradación de prestaciones mucho mayor que en un sistema con varias máquinas virtuales, debido a que en las MVs se definen los

recursos que pueden utilizar.

El impacto sufrido por las aplicaciones depende de los recursos que más requieren, por lo que la competición por ciertos recursos puede provocar una degradación de las prestaciones. Se va a tratar de detectar cómo afecta, a las prestaciones de una aplicación, cada uno de los recursos de una máquina.

- **CPU**: La unidad central de procesamiento, o CPU, se considera uno de los recursos esenciales de cualquier sistema; de manera simplificada, se puede caracterizar con la velocidad de procesamiento y con el número de núcleos. Las MV se despliegan con unos parámetros concretos en cuanto a CPU, esto es, el número de núcleos que va a tener disponibles. Estos núcleos son utilizados en exclusiva por dicha máquina. Los contenedores, por su parte, pueden ser desplegados indicando el límite de CPU, comportándose como una MV en cuanto a la eficiencia de uso de recursos. Sin embargo, más interesante supone que utilicen toda la CPU disponible dando unas garantías mínimas. Esto provoca que, como se ha indicado previamente, los contenedores de una misma máquina compitan por todos los recursos de procesamiento, dependiendo las prestaciones, en última instancia, del resto de contenedores desplegados en la misma máquina. Como se comentará más adelante, Kubernetes proporciona unos mecanismos para la limitación de la CPU que son insuficientes para evitar la degradación.

- **Jerarquía de Memoria**: El despliegue de máquinas virtuales permite limitar el tamaño de la memoria RAM. En cambio, el resto de recursos de la jerarquía de memoria, como el ancho de banda, se comparten tanto en máquinas virtuales como en contenedores desplegados sobre una misma máquina. En [27] se analiza este problema para MVs; sin embargo, las implicaciones en sistemas de contenedores donde el aislamiento general es menor no se han estudiado.

- **Disco**: Las tecnologías de máquinas virtuales pueden limitar la capacidad de disco de la máquina virtual pero el acceso se realiza a través del mismo soporte físico, el propio disco de la máquina física. En cuanto a los contenedores, todos los contenedores desplegados sobre una misma máquina acceden al mismo disco, lo que implica que si hay dos aplicaciones al mismo tiempo que realizan muchos accesos a disco, van a molestarse entre ellas provocando una degradación de prestaciones.

- **Red**: El uso de la red es último de los recursos que se analizan en esta sección, aunque su análisis queda fuera del alcance de este proyecto. En muchas configuraciones, las máquinas virtuales acceden a la red a través de su *host* lo que hace que si hay muchas máquinas virtuales sobre una misma máquina, se comparta el ancho de banda de la red de la máquina *host* provocando latencia en sus peticiones. El concepto de *pod* de Kubernetes añade un nivel de indirección adicional. Todos los contenedores dentro de un *pod* comparten una misma IP, lo que implica una posible degradación en las presentaciones. Existen algunas herramientas como tc [37], que permite aislar flujos según la máquina virtual.

Capítulo 4

Evaluación de la degradación de prestaciones experimentalmente

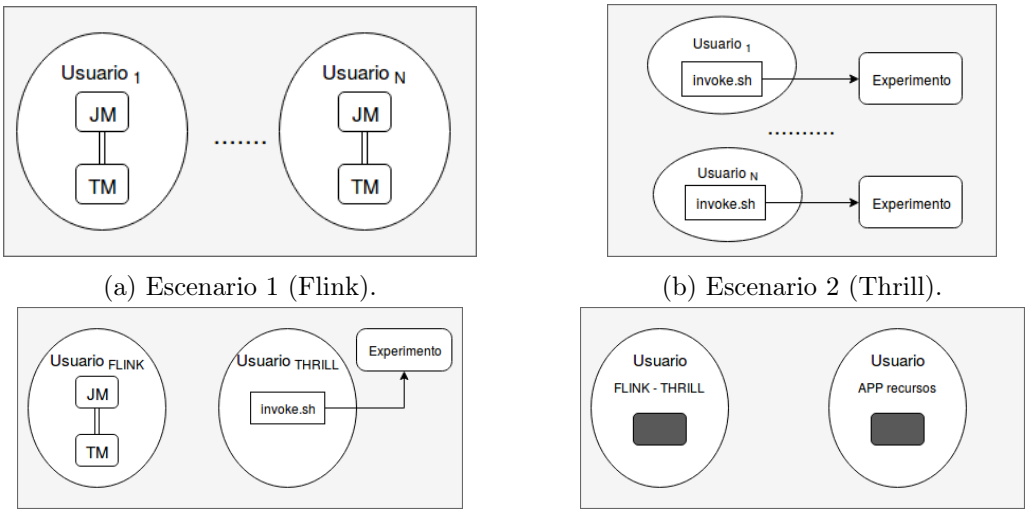
El objetivo central consiste en probar empíricamente la existencia en la degradación de prestaciones al ejecutar en una misma máquina física diferentes contenedores. Para ello, se va a presentar un entorno experimental sobre el que se plantean una serie de escenarios en los que se ejecutarán un conjunto de experimentos. Se va a tomar el tiempo de ejecución como base para realizar la comparación en las distintas situaciones.

4.1. Entorno experimental

El sistema sobre el que se ejecutan los experimentos se compone de dos máquinas que forman un *cluster* de Kubernetes (versión 1.5.2). Ambas máquinas tienen dos núcleos (Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz) y 8.0 GiB de memoria RAM. Además, ambas han sido gestionadas a través de MAAS, un sistema de provisionamiento de *bare metal*, desplegadas con una versión de Ubuntu 16.04 LTS. Todos los experimentos se van a realizar desplegando *pods* sobre la misma máquina, debido a que es sobre una única máquina física donde se puede apreciar de forma clara la degradación.

4.1.1. Escenarios

Se va a plantear cuatro escenarios sobre los cuales se realizan una serie de experimentos. Sobre cada uno de ellos se prueban diferentes situaciones que van a ser comparadas con el fin de encontrar una degradación en todas ellas. En todos los escenarios, se ejecutan las mismas aplicaciones de procesado de *data streaming* y se obtienen una serie



(a) Escenario 1 (Flink).

(b) Escenario 2 (Thrill).

(c) Escenario 3 (Flink + Thrill).

(d) Escenario 4 (Flink | Thrill + Apps).

Figura 4.1: Escenarios de ejecución de los experimentos.

de datos para posteriormente realizar una comprobación de las diferentes situaciones y demostrar que existe una degradación contra más aplicaciones de distintos usuarios estén desplegadas al mismo tiempo.

- **Escenario 1. Flink**: Sobre este escenario se despliega una aplicación Flink (con la imagen Docker explicada en el Anexo B.3) para cada uno de los usuarios sobre la que se manda un trabajo a su Job-Manager. Esta ejecución se realiza a través del comando *run* de Flink sobre cada uno de Job-Managers. Como se ve en la Figura 4.1a, sobre un mismo *host* se despliegan un Job-Manager y su Task-Manager y cada uno utiliza todos los task-slots del Task-Manager (Todos los núcleos de la máquina *host*).

- **Escenario 2. Thrill**: En el escenario Thrill, similar al de Flink, se despliega un manifiesto con la imagen Docker (el proceso detallado se explica en el Anexo C.3) y en el mismo se ejecuta cada uno de los experimentos. En la Figura 4.1b se muestra como se ejecutan los experimentos de Thrill sobre el sistema de experimentación. Se crea un contenedor de Thrill, sobre el cual se ejecuta *invoke.sh* y este utiliza todo el *host* para dicha ejecución. Además, se puede observar la posibilidad de convivencia entre distintos usuarios utilizando Thrill.

- **Escenario 3. Escenario mixto**: El tercer escenario de pruebas es el que se juntan aplicaciones de Thrill y de Flink sobre el mismo *host*. En estos experimentos se plantea la posibilidad de que varios usuarios utilicen diferentes *frameworks* de procesado de *data streaming* sobre la misma máquina. Las imágenes de Docker utilizadas son las mismas que en los escenarios 1 y 2; y el modo de ejecutar cada trabajo también es similar. Los usuarios Flink y Thrill son similares a los de los

escenarios 1 y 2 respectivamente.

- **Escenario 4. Escenario límite**: En el último escenario de experimentación se ejecuta un aplicación de procesado de *data streaming* junto con otras aplicaciones que realicen un consumo elevado de alguno de los recursos del sistema (CPU, acceso a disco o uso de RAM). En la Figura 4.1d se representa la ejecución de una aplicación de procesado de *data streaming* (no se sabe cual, desde el punto de vista del gestor de *cluster* los *pods* son como cajas negras) y otro usuario ejecutando otro tipo de aplicación (en este caso, alguna que consuma recursos).

4.2. Experimentos

En esta sección se definen los experimentos que se van a ejecutar sobre los escenarios presentados. El objetivo es demostrar que, sobre una misma máquina en la que hay varios contenedores desplegados, existe una degradación en las prestaciones de las aplicaciones que se ejecutan en esos contenedores.

Se utilizan dos *frameworks* distintos de procesado de *data streaming*, como son Flink (Anexo B) y Thrill (Anexo C), para demostrar la hipótesis con dos herramientas distintas, implementadas con diferentes lenguajes de programación (JAVA y C++ respectivamente). Sobre estos *frameworks* se van a ejecutar dos aplicaciones industriales de referencia en *MapReduce* y en procesado de grados, `WordCount` y `PageRank` respectivamente, ambas disponibles en los ejemplos de los dos *frameworks* y con el mismo algoritmo.

Se van a ejecutar dos conjuntos de experimentos. El primero tiene como objetivo medir la degradación de prestaciones al ejecutar aplicaciones cuando en la misma máquina se ha desplegado una aplicación que usa continuamente un recurso determinado. En cambio, el segundo conjunto, pretende mostrar la degradación cuando se están ejecutando aplicaciones reales. Se va a medir la degradación de prestaciones sobre el tiempo de ejecución ya que es la métrica más sencilla que mayor impacto tiene para el usuario que despliega aplicaciones.

Para todos los experimentos se han utilizado los mismos ficheros de prueba en ambos *frameworks*. Para el caso de `WordCount` se utilizan cuatro ficheros de uno, veinte, cien y mil millones de palabras; en el caso de `PageRank` son ficheros de mil, diez mil, cien mil y un millón de nodos, además de un fichero que representa un grafo de más de 300000 nodos. La distribución del grado de los nodos de los grafos sigue una distribución *power law* [38]. Se han construido utilizando el algoritmo de Barabasi-Albert de la biblioteca Networkx[1], implementada en python. La Figura 4.1 muestra el tamaño de los ficheros empleados.

[1] `https://networkx.github.io/`

Fichero	Tamaño del Fichero (MB)
WC 1millon pal	6,612
WC 20millones pal	132,246
WC 100millones pal	661,231
WC 1000millones pal	6.612,310
PR 1000 nodos	0,04
PR 10000 nodos	0,45
PR 100000 nodos	5,42
PR 334863 nodos	12,29
PR 1000000 nodos	63,87

Tabla 4.1: Tabla de tamaños de los ficheros de prueba.

Aplicación	Recurso	Indicaciones
Pov-ray [39]	CPU	Version 3.7 con paralelismo por defecto
Stream [40]	Memory Bandwidth	-DSTREAM_ARRAY_SIZE=100000000 -DNTIMES=100
dd [41]	Disk I/O Bandwidth	dd if=/dev/zero of=/root/testfile bs=1G count=1 oflag=direct > dev/null

Tabla 4.2: Aplicaciones de referencia utilizadas como trabajo de fondo, con el recurso que usan de forma intensiva y sus parámetros de ejecución.

Todos los experimentos que se muestran a continuación se han ejecutado diez veces, tomando todos los valores y calculando la media. Para calcular la degradación de prestaciones, primero se toma como referencia el tiempo de ejecución de la aplicación en solitario (por ejemplo App_0). Después se mide el tiempo de ejecución al ejecutarse otras en la misma máquina (siguiendo el ejemplo con Pov-ray sería App_{pv}) y se calcula el cociente ($\frac{App_{pv}}{App_0}$).

4.2.1. Aplicaciones de referencia

El primer conjunto de experimentos consiste en la ejecución de las aplicaciones de *data streaming* que se han comentado anteriormente, con aplicaciones de referencia que hacen un uso intensivo de un recurso, explicadas en la Tabla 4.2. Estas aplicaciones se ejecutan con los parámetros indicados en la tabla, en un bucle continuo.

Los resultados se muestran en la Figura 4.2. Lo primero que se observa es que la degradación depende del tamaño de la entrada. En el caso de ejecutar `WordCount` con el fichero más grande a la vez que `dd`, se produce un grave impacto debido a que el tamaño de dicho fichero implica el acceso a disco de forma continuada (los tiempos de ejecución son, aproximadamente, cuatro veces peores). Esto ocurre para ambos *frameworks*.

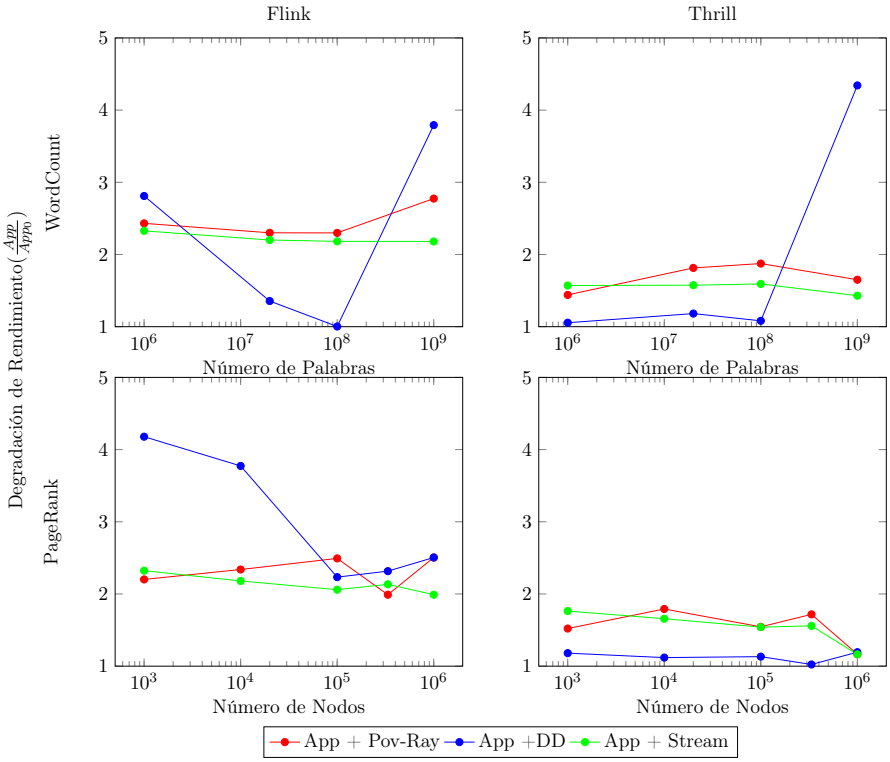


Figura 4.2: Degradación de rendimiento para diferentes aplicaciones. `WordCount` sobre Flink, `WordCount` sobre Thrill, `PageRank` sobre Flink y `PageRank` sobre Thrill con aplicaciones de uso intensivo de recursos de fondo.

En segundo lugar, se puede observar que para ambos *frameworks*, la degradación de `PageRank` es menor que la de `WordCount`. Además, se puede observar como el *framework* Thrill tiene una menor degradación al juntarlo con otras aplicaciones que Flink.

Con estos datos queda demostrada la hipótesis planteada de la degradación de rendimiento en el momento que varios contenedores se despliegan sobre la misma máquina física.

4.2.2. Aplicaciones industriales

El segundo conjunto de experimentos se presentan en la Figura 4.3 y muestran la degradación de rendimiento en situaciones donde se juntan diferentes aplicaciones reales. Se han realizado experimentos uniendo dos Flink – Thrill – y cuatro Flink – Thrill – al mismo tiempo. Se han completado con experimentos donde se ejecutan simultáneamente ambos *framework* (con una y dos instancias de cada uno).

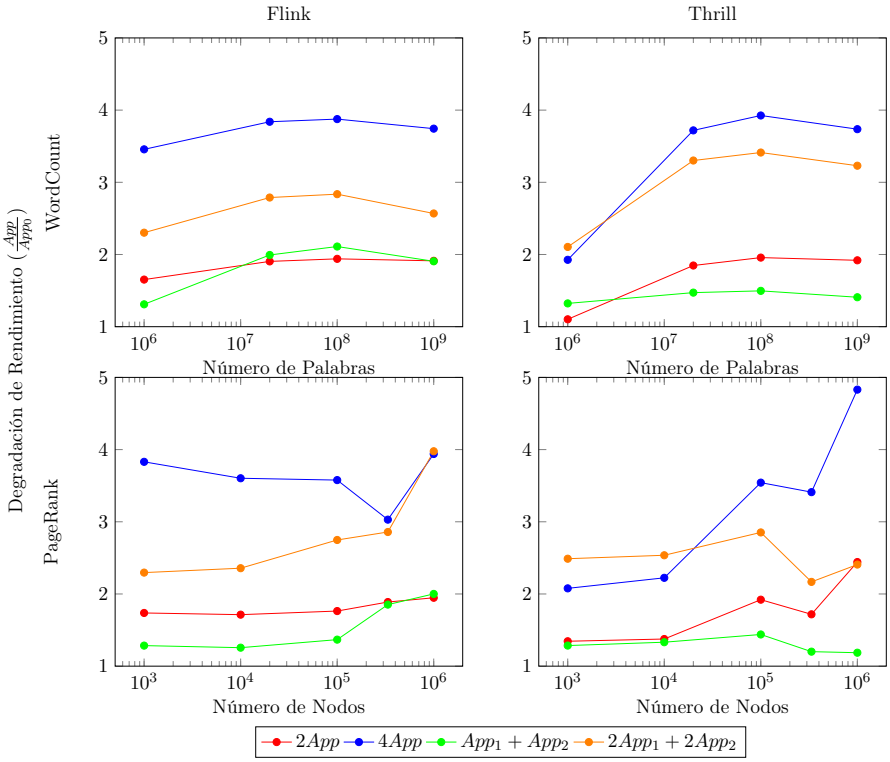


Figura 4.3: Degradación de rendimiento respecto a la ejecución en solitario.

Tal y como cabría esperar se produce una degradación de prestaciones. Se puede observar como el impacto es similar para la mayoría de situaciones y como la situación de cuatro aplicaciones al mismo tiempo es la de mayor degradación. Sin embargo, en el caso de `PageRank` sobre Flink sufre un mayor impacto en dos Flink + dos Thrill en el fichero más grande que con cuatro Flink, aunque la diferencia es mínima.

Capítulo 5

Scheduler de cliente

Partiendo de la degradación de prestaciones detectada en el Capítulo 4, en este capítulo se desarrolla una solución que mejore el rendimiento del *cluster*. Ha sido desarrollada en python y se muestra en el Anexo D. Como mecanismo se propone acercar el *scheduler* al usuario planteando la posibilidad de que el mismo caracterice, a través de etiquetas informales, la aplicación que va a desplegar en el *cluster*. De esta manera se sigue manteniendo la transparencia en el provisionamiento. El *scheduler* tratará de separar aplicaciones que usen el mismo recurso. De esta forma, se da respuesta a los dos problemas encontrados durante la fase de hipótesis y experimentación.

5.1. Diseño de la Arquitectura

La solución propuesta implica añadir un nuevo componente en la arquitectura del sistema. Este componente traduce la caracterización efecutada por el usuario a los mecanismos base del *scheduler* de Kubernetes, principalmente el mecanismo de *labels*. La arquitectura modificada se muestra en la Figura 5.1. Con esta solución los usuarios acceden al *cluster* a través del nuevo componente que obtiene el estado del sistema y elige la máquina destino para cada aplicación según su caracterización y las aplicaciones que hay en el *cluster*.

5.2. Caracterización de aplicaciones

Para ilustrar esta metodología, se van a caracterizar las aplicaciones que han sido utilizadas en la experimentación. En la Figura 5.2 se muestra una gráfica en la que sitúan las 18 aplicaciones utilizadas, junto con una tabla que indica su caracterización. Cada

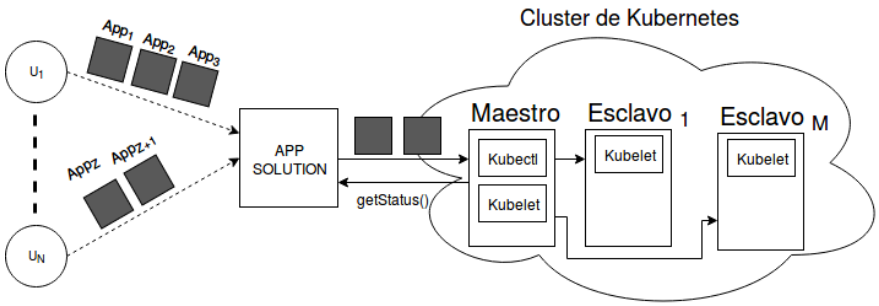


Figura 5.1: Representación de la nueva Arquitectura del sistema.

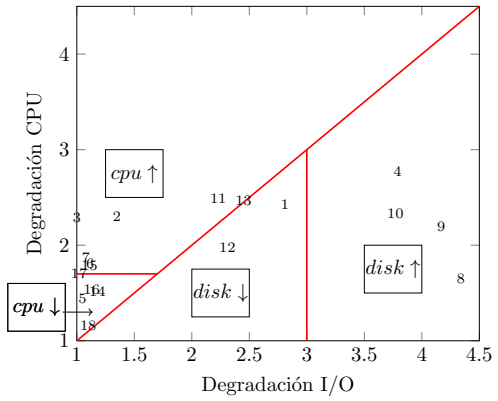
aplicación se sitúa respecto a su degradación en caso de ejecutarse con una aplicación de uso intensivo de disco (eje de ordenadas) y de uso intensivo de CPU (eje de abscisas). Se ha separado de forma cualitativa las aplicaciones según el uso de recursos que hacen, siendo un 3 el valor que separa el uso alto y bajo de disco y 1.7 el valor que separa el uso alto y bajo de CPU. Se puede observar cuales son las aplicaciones que sufren una mayor degradación a través de la relación entre la degradación con una aplicación de uso intensivo de disco y la degradación con una de uso intensivo de CPU.

En este caso, por sencillez, se han tenido en cuenta dos recursos (cpu y disco) y dos niveles de uso (alto y bajo). Esta caracterización es una ilustración sencilla de la metodología utilizada, que puede hacerse más compleja a través de la introducción de más recursos, más niveles de uso o mecanismos de clasificación automáticos.

Como se ha señalado anteriormente, la caracterización debe ser específica para cada tamaño de entrada, ya que afecta de forma distinta. En esta línea, se puede observar el caso de Flink con `WordCount`, en el que la aplicación `dd` afecta menos en el caso del segundo y tercer fichero de entrada y provoca un gran impacto en el último experimento. Esto es debido al tamaño del fichero (mostrado en la Figura 4.1), ya que el último fichero no se almacena entero en RAM y provoca fallos de lectura teniendo que acceder a disco en todo momento.

5.3. Diseño del *scheduler*

A partir de la evidencia experimental presentada en la Capítulo 4, se proponen dos criterios para decidir el destino del contenedor. La primera idea consiste en minimizar el número de máquinas sin utilizar. También, se buscará que el número de contenedores en cada máquina esté lo más balanceado posible, evitando que en una máquina haya muchas aplicaciones y en otra una o ninguna. Como el número de aplicaciones no tiene que ser un múltiplo del número de máquinas, no siempre será posible balancear.



(a) Relación entre degradación con dd y con Pov-ray.

App	Id	Categoría
FlinkWC 1millon	1	*cpu* ↑
FlinkWC 20millones	2	*cpu* ↑
FlinkWC 100millones	3	*cpu* ↑
FlinkWC 1000millones	4	*disk* ↑
ThrillWC 1millon	5	*cpu* ↓
ThrillWC 20millones	6	*cpu* ↑
ThrillWC 100millones	7	*cpu* ↑
ThrillWC 1000millones	8	*disk* ↑
FlinkPR 1000	9	*disk* ↑
FlinkPR 10000	10	*disk* ↑
FlinkPR 100000	11	*cpu* ↑
FlinkPR 334863	12	*disk* ↓
FlinkPR 1millon	13	*cpu* ↑
ThrillPR 1000	14	*cpu* ↓
ThrillPR 10000	15	*cpu* ↑
ThrillPR 100000	16	*cpu* ↓
ThrillPR 334863	17	*cpu* ↑
ThrillPR 1millon	18	*cpu* ↓

(b) Resultado de la caracterización de las aplicaciones.

Figura 5.2: Caracterización de las aplicaciones utilizadas en la experimentación. La columna Id de la tabla corresponde con el identificador de la aplicación del gráfico.

El segundo criterio consiste en separar aplicaciones que utilicen el mismo recurso. Para ello, dada una aplicación caracterizada por el usuario, se calcula un valor para cada máquina destino, que indicará la penalización que provocaría desplegarla en esa máquina. El menor valor indicará la máquina en la cual el impacto es menor.

La caracterización que realizan los usuarios de sus aplicaciones a través de etiquetas en la primera línea del manifiesto de Kubernetes, se traducen a etiquetas de *nodeSelector* para la elección de la máquina destino. Para almacenar la información sobre las aplicaciones lanzadas al *cluster*, se va a utilizar un fichero que guardará una aplicación por fila (nombre, máquina sobre la que está desplegada y las etiquetas que le asignó el usuario).

De manera informal el *scheduler* de alto nivel tiene cinco pasos:

1. Obtención del estado actual del *cluster* de Kubernetes con el fin de conocer las aplicaciones que siguen en ejecución.

2. Actualización del fichero de estado del sistema, quitando las aplicaciones que hayan terminado y no estén ejecutándose en el *cluster*.

3. Extracción de la etiqueta de la aplicación que el usuario ha añadido al manifiesto y eliminación de esa línea para crear un manifiesto válido.

4. Decisión de la maquina destino y modificación del manifiesto. Se elige aquella en la que se causa menor penalización

5. Lanzamiento de la aplicación al *cluster* y guardado de su información en el fichero de estado del sistema.

Algorithm 1 Client-Side Scheduler

```
1: procedure Client-Side Scheduler(l_app, W)
2:     S = GetClusterState()
3:     minValue := ∞
4:     bestNode := 0
5:     for N in S do
6:         if |N| ≤ min{|M| ∈ S}) then
7:             if minValue > Σ_j^|N| w_{j,app} then
8:                 minValue := Σ_j^|N| w_{j,app}
9:                 bestNode := N
10:            end if
11:        end if
12:    end for
13:    Allocate(l_app, bestNode)
14: end procedure
```

$App_1 \setminus App_2$	$cpu\uparrow$	$cpu\downarrow$	$disk\uparrow$	$disk\downarrow$
$cpu\uparrow$	5	4	2	1
$cpu\downarrow$	4	3	1	0
$disk\uparrow$	2	1	5	4
$disk\downarrow$	1	0	4	3

Tabla 5.1: Matriz de pesos W para dos recursos y dos niveles de uso. Las categorias son: uso alto de CPU– $cpu\uparrow$ –, uso bajo de CPU – $cpu\downarrow$ –, uso alto de I/O – $disk\uparrow$ –, y uso bajo de I/O – $disk\downarrow$.

Formalmente, se define un nodo N como un multi-conjunto de etiquetas. Cada etiqueta representa la aplicación que está desplegada en ese nodo. En este caso, hay cuatro etiquetas – l_0 equivalente a $cpu\uparrow$, l_1 equivalente a $cpu\downarrow$, etc. En un momento concreto, el estado del *cluster* $\boldsymbol{S}$ puede ser modelado como un conjunto de nodos. Dada una nueva aplicación, con una etiqueta l_{app} el mejor nodo para desplegarla es dado por:

$$\underset{i\in 0}{\operatorname{argmin}}^{|E|} \sum_{j} w_{E_{i,j},app}$$

donde $w_{i,j}$ es el peso de i-ésima fila y j-ésima columna de una matriz de pesos W (definida en la Tabla 5.1) y $E_{i,j}$ es la j-ésima etiqueta de la aplicación de i-ésima conjunto de nodos E. E es definido como $E = \{N \in \boldsymbol{S} \wedge \forall M \in \boldsymbol{S}, |N| \leq |M|\}$. El conjunto E contiene los nodos con menos aplicaciones. La formalización previa puede ser implementada en un algoritmo, como se muestra en el Algoritmo 1.

A través de los experimentos de la Sección 4, se puede inferir el proceso de construc-

ción de una matriz de pesos W. Cada posición de la matriz w_{ij} indica la penalización de desplegar una aplicación con etiqueta j sobre un nodo donde hay una aplicación con etiqueta i. Se construye teniendo en cuenta si utilizan el mismo recurso. Si así es, se establecen los niveles más altos de penalización (5, 4 y 3) según el nivel de uso del recurso. Si ambas tiene un nivel de uso alto se establece la penalización mayor (5), si uno es bajo y otro alto (4) y si ambos son bajos (3). En el caso de que sean recursos distintos, se realiza el mismo proceso pero con los valores de menor penalización (2, 1 y 0).

5.4. Implementación de la solución

La implementación del Algoritmo 1 se incluye en el Anexo D y se ha desarrollado en python. La función *GetClusterState* corresponde con la función *getStatus* y se encarga de obtener los *pods* que están en ese momento ejecutándose en el *cluster*. Este estado se obtiene en primer lugar cuando se va a lanzar una aplicación a través de la solución. Una vez analizados los *pods* que hay ejecutándose y actualizado el fichero de estado del *cluster* (función *deleteApps*), se elige el nodo destino (función *decideMachine*) como en el algoritmo explicado.

Tras modificar el manifiesto Kubernetes, a través de las funciones *deleteLine* (para eliminar la línea de etiquetas informales) y *addNodeSelector* (para añadir el nodo destino), y modificar el fichero de estado del *cluster* (función *saveUserApp*) se lanza la aplicación al *cluster*, correspondiente con la función *Allocate* del algoritmo.

5.5. Validación experimental del *scheduler*

El sistema de experimentación para estos escenarios se compone de un *cluster* Kubernetes con 8 máquina de 4 núcleos (Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz) y 8 GB de RAM cada una. Sobre el primer nodo se despliega el maestro del *cluster* y el resto de nodos serán los esclavos.

En esta sección, se va a probar que la solución planteada, provoca una mejora en el rendimiento de las aplicaciones dentro del sistema. Se van a desplegar un total de 18 aplicaciones, 3 `dd`, 3 `Pov-ray`, 3 `WordCount` con Flink, 3 `PageRank` con Flink, 3 `WordCount` con Thrill y 3 `PageRank` con Thrill; en estos últimos casos se van a emplear los ficheros de entrada más grandes.

El primer conjunto de experimentos consiste en el lanzamiento de estas 18 aplicaciones de forma simultánea en el *cluster* con el *scheduler* por defecto de Kubernetes (Figura 5.3) y se compara con resultado de desplegarlas a través del *scheduler* propuesto (Figura 5.4).

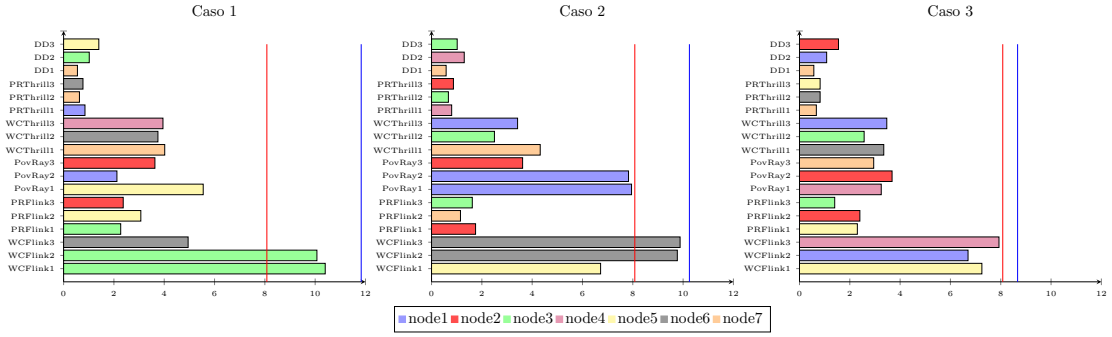


Figura 5.3: Tiempo de ejecución y nodo asignado por el *scheduler* por defecto de Kubernetes. La línea azul representa el tiempo total (T creación + T ejecución + T finalización de los *pods*), línea roja es el tiempo del *scheduler* de cliente propuesto.

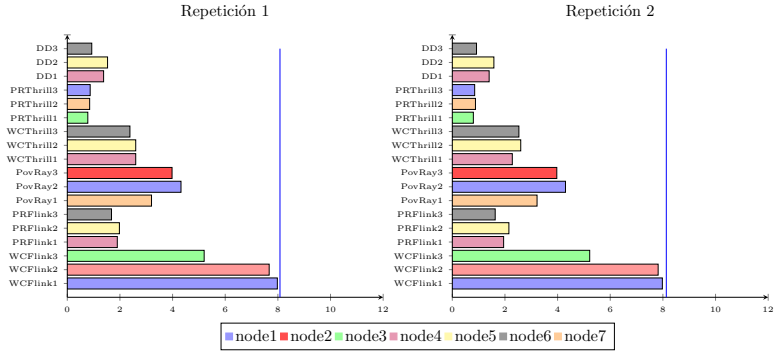


Figura 5.4: Tiempo de ejecución y nodo asignado por el *scheduler* propuesto. La línea azul indica el tiempo total de ejecución (T creación + T ejecución + T finalización de los *pods*).

En la Figura 5.3 se han elegido tres casos significativos (entre 10) para poner de manifiesto los problemas que pueden surgir. Se puede comprobar que no es determinista, ya que cada vez despliega las aplicaciones en diferentes nodos, lo que provoca una gran variabilidad en los resultados (de casi 12 minutos de tiempo de ejecución total en el caso 1 a los poco más de 8 en el caso 3, un mejor caso que no mejora el resultado del *scheduler* de cliente que se plantea como solución). Esto implica que no se pueda medir el impacto de las aplicaciones sobre el sistema, ni el rendimiento que puede tener cada una de las aplicaciones.

Además, en los casos dos y tres se puede observar que este sistema no balancea las aplicaciones, colocando cuatro contenedores en el nodo 3 y uno en el nodo 4 (Caso 1) o cuatro contenedores en el nodo 3 y uno en el nodo 5 (Caso 2). También, se puede comprobar como une dos aplicaciones `Pov-ray` (Caso 2) que tiene un grave impacto en el tiempo total de ejecución. En los tres casos se observa como el tiempo total de ejecución (línea azul) se encuentra muy por encima del resultado del *scheduler* de cliente.

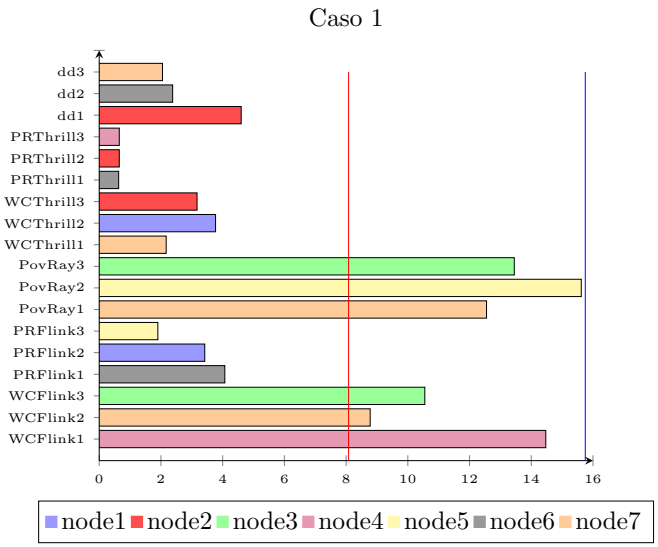


Figura 5.5: Tiempo de ejecución y nodo asignado con limitación de CPU. La línea roja indica el tiempo total de ejecución con el *scheduler* propuesto y la línea azul el tiempo total de ejecución medido (T creación + T ejecución + T finalización de los *pods*).

Para el *scheduler* de cliente propuesto, los resultados presentan poco varianza debido al carácter determinista del mismo. Además se hace un balanceo de aplicaciones para evitar situaciones de sobrecarga en un nodo. Los tiempos se mantienen estables en poco más de 8 minutos en cualquier caso. Con este nuevo sistema, se pueden garantizar al usuario ciertos parámetros de calidad de servicio. Se muestran dos repeticiones (entre 10) en la Figura 5.4 ya que todas las ejecuciones que se realicen dan resultados casi exactos.

Por último, el segundo conjunto de experimentos tienen como fin mostrar que los mecanismos propios de Kubernetes, como son las etiquetas *limits* y *request*, no son suficientes para lidiar con el problema señalado. Estas etiquetas permiten reservar una serie de recursos para el *pod* que se despliega lo que implica acercarse al concepto de máquina virtual con sus recursos definidos. Aunque se definan los *limits* y *request*, algunos otros recursos, como el acceso a disco no se puede aislar. Otro problema que surge a la hora de limitar recursos es que se deja algunos de los recursos de las máquinas sin utilizar ya que se reservan todos los recursos aunque la aplicación no lo necesite.

En el experimento, se ha planteado un límite de 1CPU y 1GB de RAM para todas las aplicaciones, excepto para Flink `WordCount`, que requiere más CPU. En la Figura 5.5 se muestra un caso (entre 10) en el que al limitar los recursos aumenta el tiempo de ejecución respecto a la solución. Estos mecanismos producen una alta variabilidad en distintas instancias de la misma aplicación, aunque tengan el mismo número de recursos reservados. Por ejemplo, `Pov-ray1` tarda poco más de 12 minutos mientras que Pov-ray3

tarda casi 16 minutos. En definitiva, se comprueba que estableciendo límites de recursos a las aplicaciones, el rendimiento global disminiye más que si todas se hubiesen ejecutado sin ellos.

El *scheduler* que se ha propuesto permite una mejora en el tiempo de ejecución de este conjunto de experimentos, de un 20 % en el caso medio y un 32 % para el peor caso del *scheduler* de Kubernetes.

Capítulo 6

Conclusiones

La virtualización a través de contenedores constituye un mecanismo más rápido y flexible que las MVs. En el presente Trabajo Fin de Grado se ha partido de la hipótesis de que la compartición de recursos entre distintos contenedores sobre una misma máquina física, puede provocar una degradación en las prestaciones debido a la falta de aislamiento. Con el fin de probar dicha hipótesis, se ha diseñado un conjunto de experimentos basado en aplicaciones industriales de *data streaming*.

Adicionalmente, se ha comprobado que los mecanismos de Kubernetes para aislar recursos, como las etiquetas *limits* y *request*, no son suficientes para lidiar dicha degradación. Aunque la CPU es el recurso que puede provocar una mayor degradación en las aplicaciones, existen otros recursos que no se pueden aislar y que causan degradación (por ejemplo, ancho de red o ancho de memoria). Además, estos mecanismos de reserva pueden dejar recursos sin utilizar en el *cluster*, provocando un importante impacto en el rendimiento de las aplicaciones.

La posible degradación debe ser tenida en cuenta por los usuarios en el momento del despliegue de sus aplicaciones. Esta degradación provoca una variabilidad en el tiempo de ejecución, lo que puede implicar que los resultados sean inviables. Detectados estos problemas, se ha planteado y desarrollado una solución para acercar el proceso de *scheduling* al usuario, permitiéndole tener un mayor control a la hora de desplegar una aplicación en el *cluster*.

El *scheduler* de cliente que se propone como solución permite obtener un mejor rendimiento en la ejecución de las aplicaciones. Se obtiene una mejora del 20 % en el caso medio y una mejora del 32 % en el peor caso del *scheduler* por defecto de Kubernetes. Además, con este *scheduler* el despliegue de aplicaciones es menos transparente para el usuario ya que deberán incluir información de su aplicación, como es el recurso que más utiliza y su nivel de uso.

Por último, se ha documentado la instalación y los componentes básicos de Kubernetes, centrándose en comandos de debugging. Además, se ha explicado la operativa y puesta en marcha sobre Kubernetes de dos *frameworks* de procesado de *data streaming* como son Flink y Thrill, que han sido utilizados para la demostración de la hipótesis planteada.

Como línea de trabajo futuro se plantea la mejora del *scheduler* propuesto, para añadir la posibilidad del provisionamiento dinámico y automático de máquinas físicas para dar respuesta a las necesidades de cada momento. Es decir, desplegar y añadir nodos al *cluster* en los momentos donde fuese necesario. En la misma línea de mejora del sistema propuesto, podrían introducirse mecanismos de clasificación de aplicaciones de forma automática y más sofisticados. Esto implicaría considerar más recursos, como el uso de ancho de banda de memoria o ancho de banda de red, y más grados de uso. Este trabajo requeriría de un análisis detallado de las aplicaciones para realizar una clasificación lo más fiable posible y para comprobar que, añadiendo más recursos o niveles de uso, se produce una mejora en el rendimiento.

Finalmente, se podría ampliar el *scheduler* de cliente para dar un servicio multiplataforma, permitiendo al usuario elegir entre distintos *clusters* federados o proveedores (por ejemplo, Kubernetes y DockerSwarm)

Bibliografía

[1] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[2] Victor Marmol, Rohit Jnagal, and Tim Hockin. Networking in containers and container clusters. *Proceedings of netdev 0.1*, 2015.

[3] Ann Mary Joy. Performance comparison between linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, pages 342–346. IEEE, 2015.

[4] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[5] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.

[6] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. Qos-aware admission control in heterogeneous datacenters. In *ICAC*, pages 291–296, 2013.

[7] Linux containers organization. `https://linuxcontainers.org/`. Accessed: 2017-05-08.

[8] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *IJCSNS*, 17(3):228, 2017.

[9] Ryan Chamberlain and Jennifer Schommer. Using docker to support reproducible research. *DOI: http://dx. doi. org/10.6084/m9. figshare*, 1101910, 2014.

[10] Openvz virtuozzo containers. Accessed: 2017-06-10.

[11] Freebsd jails. Accessed: 2017-06-10.

[12] Solaris containers. Accessed: 2017-06-10.

[13] Rocket container. `https://github.com/rkt/rkt/blob/master/Documentation/app-container.md`. Accessed: 2017-05-08.

[14] Steve G Langer and Todd French. Virtual machine performance benchmarking. *Journal of digital imaging*, 24(5):883–889, 2011.

[15] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66(105-111):2, 2014.

[16] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186, 2013.

[17] Docker swarm overview. `https://docs.docker.com/swarm/overview/`. Accessed: 2017-05-08.

[18] Openshift. `https://www.redhat.com/es/technologies/cloud-computing/openshift`. Accessed: 2017-05-14.

[19] João Gama and Pedro Pereira Rodrigues. Data stream processing. In *Learning from Data Streams*, pages 25–39. Springer, 2007.

[20] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. *USENIX Login*, 37(4):45–51, 2012.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[23] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[24] Lambda architecture. `http://lambda-architecture.net/`. Accessed: 2017-05-16.

[25] Norman Bobroff, Andrzej Kochut, and Kirk Beaty. Dynamic placement of virtual machines for managing sla violations. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 119–128. IEEE, 2007.

[26] David Hoeflin and Paul Reeser. Quantifying the performance impact of overbooking virtualized resources. In *Communications (ICC), 2012 IEEE International Conference on*, pages 5523–5527. IEEE, 2012.

[27] Navaneeth Rameshan, Ying Liu, Leandro Navarro, and Vladimir Vlassov. Hubbub-scale: Towards reliable elastic scaling under multi-tenancy. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 233–244. IEEE, 2016.

[28] Cristian Ruiz, Emmanuel Jeanvoine, and Lucas Nussbaum. Performance evaluation of containers for hpc. In *European Conference on Parallel Processing*, pages 813–824. Springer, 2015.

[29] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172. IEEE, 2015.

[30] Moritz Raho, Alexander Spyridakis, Michele Paolino, and Daniel Raho. Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing. In *Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in*, pages 1–8. IEEE, 2015.

[31] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 386–393. IEEE, 2015.

[32] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[33] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[34] Sungmin Choi, Rohyoung Myung, Heeseok Choi, Kwangsik Chung, Joonmin Gil, and Heonchang Yu. Gpsf: General-purpose scheduling framework for container based on cloud environment. In *Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2016 IEEE International Conference on*, pages 769–772. IEEE, 2016.

[35] Aurelien Havet, Valerio Schiavoni, Pascal Felber, Maxime Colmant, Romain Rouvoy, and Christof Fetzer. Genpack: A generational scheduler for cloud data centers. In *Cloud Engineering (IC2E), 2017 IEEE International Conference on*, pages 95–104. IEEE, 2017.

[36] Kubernetes scheduler. `https://kubernetes.io/docs/admin/kube-scheduler/`. Accessed: 2017-05-07.

[37] Linux traffic control. `http://tcng.sourceforge.net/doc/tcng-overview.pdf`. Accessed: 2017-05-03.

[38] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM computer communication review*, volume 29, pages 251–262. ACM, 1999.

[39] Persistence of Vision Pty. Ltd. (2004). Persistence of vision raytracer (version 3.7) [computer software].

[40] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[41] dd(1) linux user's manual.

Anexos

Anexo A

Despliegue de Kubernetes

Los pasos necesarios para realizar una correcta instalación de Kubernetes no están enteramente recogidos en la documentación en `kubernetes.io`. Es por ello que aquí se recogen mostrando los posibles problemas y sus soluciones así como alternativas en la configuración de la instalación.

A.1. Requisitos previos

Se deben cumplir los siguientes requisitos antes de comenzar con el proceso de instalación:

- Al menos una máquina con SO Ubuntu 16.04, CentOS 7 o Hypriotv1.0.1 o versiones superiores. A pesar de que con una máquina pueden funcionar algunas aplicaciones pequeñas se recomienda trabajar con 3 máquinas para que una trabaje como *Master* y 2 como *Esclavos*.
- Al menos 1 GB de RAM para cada una de las máquinas. Para ciertas aplicaciones puede ser necesario tener una mayor memoria RAM, como aplicaciones JAVA.
- Total conectividad entre las máquinas, ya sea con una red pública o con una privada.

A.2. Instrucciones

A.2.1. Instalación de los paquetes necesarios

Los paquetes que deben instalarse se indican en la Tabla A.1.

Docker	Sistema de contenedores que utiliza Kubernetes a bajo nivel. Se recomienda utilizarlo solamente si Kubernetes no tiene mecanismos similares para ciertas funcionalidades o para realizar debugging.
Kubelet	Sistema core del sistema Kubernetes. Kubectl opera en todas las máquinas del *cluster* y realiza funciones muy importantes como el manejo de *pods*.
Kubectl	Comando para realizar control es el *cluster*. Es recomendable tenerlo en todas las máquinas aunque solo es estrictamente necesario en el maestro.
Kubeadm	Este comando permite realizar la instalación del *cluster*, se utiliza tanto en el maestro como en los esclavos como se va a ver a continuación.

Tabla A.1: Paquetes necesarios en la instalación de Kubernetes

Antes de explicar como se deben instalar estos paquetes indicar que se debe ser superusuario *(root)* en el sistema.

Instalación en Ubuntu o HypriotOS

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
$ cat "<<EOF > /etc/apt/sources.list.d/kubernetes.list"
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
$ apt-get update

# Instalar docker si no se tiene instalado ya.
$ apt-get install -y docker.io

# Instalación paquetes de Kubernetes.
$ apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

Instalación en CentOS

```
$ cat "<<EOF > /etc/yum.repos.d/kubernetes.repo"
[kubernetes]
name=Kubernetes
baseurl=http://yum.kubernetes.io/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
# Deshabilitar SELinux
$ setenforce 0
$ yum install -y docker kubelet kubeadm kubectl kubernetes-cni
$ systemctl enable docker && systemctl start docker
$ systemctl enable kubelet && systemctl start kubelet
```

NOTA: Se deshabilita SELinux para permitir el acceso de los contenedores al sistema de ficheros. Kubernetes todavía no soporta SELinux de forma correcta.

A.2.2. Instalación del nodo maestro

Una vez se han instalado todos los paquetes de la sección anterior, se debe seleccionar una de las máquinas para ser el maestro del *cluster*. El maestro es la máquina donde va a operar el panel de control del *cluster*, como *etcd* (que es la base de datos del *cluster*) y el *servidor API* (con el cual se comunica **kubectl**). Todos estos componentes operan sobre *pods* que lanza **kubelet**.

El comando para lanzar y crear todos los componentes necesarios es **kubeadm init** pero hay que tener en cuenta ciertos aspectos antes de utilizarlo.

Flags de kubeadm init

El primer aspecto a tener en cuenta son los distintos *flags* que existen en el comando **kubeadm init**, los más interesantes son:

- ***–api-advertise-addresses*** *<**ip-address**>*: **kubeadm init** por defecto autodetecta la interfaz de red en la que anunciar el maestro. Con este *flag* se puede especificar la red a utilizar.

- ***–skip-preflight-checks***: **kubeadm** ejecuta una serie de comprobaciones previas a la instalación, algunos de ellos serán solo avisos pero otros provocarán la salida del kubeadm hasta que se solucione el problema. A través de este *flag* se pueden evitar algunas de esta comprobaciones previas.

  NOTA: Algunos de los errores seguirán apareciendo a pesar del flag *por lo que habrá que solucionarlo a mano o utilizando un comando de reset (**kubeadm reset**, que se explicará más adelante).*

- ***–token***: kubeadm utiliza un token para realizar la conexión entre los esclavos y el maestro. Por defecto se genera uno pero con este *flag* se puede indicar un token específico.

- Otros *flags* como ***–service-cidr***, que es para sobreescribir la subred que utiliza Kubernetes (por defecto 10.96.0.0/12) o ***–use-kubernetes-version***, para elegir la versión de Kubernetes a utilizar, son interesantes para casos más concretos pero no muy necesarias en la mayoría de los casos.

Teniendo claro las posibilidades que proporciona **kubeadm init** ya se puede ejecutar. Durante la ejecución de este comando se van a crear de los ficheros de configuración, el panel de control, la API del cliente, etc. Se debe guardar el **token** que se proporciona al inicio del comando, ya que se utilizará para unir nodos al *cluster*.

NOTA: Por defecto, Kubernetes no selecciona el maestro para despliegue de pods *por motivos de seguridad. Si se quiere que el maestro pueda desplegar* pods *hay que usar **kubectl taint nodes –all dedicated-**.*

A.2.3. Instalación de un sistema de gestión de red

Es necesario instalar un sistema para gestionar la red del *cluster*, con el fin de que los *pods* puedan conectarse entre ellos. **Se debe realizar esta instalación antes de desplegar cualquier aplicación en el *cluster*.** Existen varios proyectos de sistemas de gestión de red, pero se ha elegido Weave Net.

Weave Net

La instalación de Weave Net es muy sencilla ya que se realiza como cualquier *pod*, ***kubectl apply -f https://git.io/weave-kube***. Transcurridos unos segundos se puede ver que el *pod* esté corriendo de forma correcta en todos los nodos y se desplegará también en cualquier nodo que se una al *cluster* posteriormente.

Al igual que ocurre con la configuración de kubeadm init, si se tiene instalada una version anterior de Weave Net se debe resetear (*weave reset*), eliminar cualquier cláusula

```
containers:
  - name: weave
    env:
      - name: IPALLOC_RANGE
        value: 10.0.0.0/16
```

Figura A.1: Ejemplo de configuración del manifiesto de Weave.

que se pueda haber añadido en tiempo de inicio y eliminar los ficheros de configuración (*rm /opt/cni/bin/weave-**). Finalmente se puede volver a añadir el *pod* como se ha indicado anteriormente.

Este proyecto está en continuo desarrollo por lo que podría ser interesante actualizarlo a la última versión. Para ello se debe volver a aplicar el último manifiesto de Weave (***kubectl apply -f https://git.io/weave-kube***). Una vez aplicado el manifiesto se debe reinicar cada *pod* a través de **kubectl delete** y esperar a que se reinicie.

NOTA: Se debe esperar a que se reinicie el pod *por completo y vuelva a estar ejecutandose de forma correcta antes de eliminar el siguiente, ya que se perderá el rango de IPs dando posibles IPs duplicadas en el* cluster.

Weave Net permite realizar modificaciones sobre la configuración de la red de *pods* que despliega. Para realizar los cambios se debe modificar el fichero *YAML* antes de ejecutar ***kubectl apply***. En la Figura A.1 se muestra un ejemplo del manifiesto de Weave Net. Las siguientes variables pueden modificarse:

- *IPALLOC_RANGE*: Por defecto, Weave Net utiliza en rango de direcciones IP de 10.32.0.0/12 y puede ser modificado siempre en el formato CIDR (como el mostrado por defecto 10.32.0.0/12).
- *CHECKPOINT_DISABLE*: Si se pone a 1, se deshabilita la búsqueda de nuevas versiones de Weave Net (por defecto, si que se buscan).
- *WEAVE_MTU*: Modificar el MTU de Weave Net para redes más especficas (por defecto, es 1376 Bytes).
- *WEAVE_EXPOSE_IP*: Permite indicar la dirección del gateway desde la red Weave hasta la red *host*. Útil para configurar el add-on (el cual añade funcionalidad a kubernetes) como un *pod* estático.
- *KUBE_PEERS*: Weave Net, por defecto, busca los peers del *cluster* Kubernetes a través del servidor API. Con esta opción se puede indicar los peers del *cluster* de forma concreta.

- *EXPECT_NPC*: Si se pone a 0, se deshabilita el Controlador de Politicas de Red (Network Policy Controller). Por defecto está a 1, habilitada. El *add-on* soporta el API de políticas de Kubernetes que permite aislar *pods* de forma segura a través de namespaces y labels.
- *IPALLOC_INIT*: Permite modificar el modo inicial del Manager de direcciones IP (IP Address Manager, encargado de dividir bloques de direcciones IP para asignar de forma única a los diferentes *pods*). Por defecto para consensuar entre los KUBE_PEERS.

NOTA: Como se ha indicado previamente, antes de aplicar el manifiesto se debe descargar el que se proporciona Weave Net y realizar las modificaciones de la configuración de la red de pods*. Estas modificaciones no deben contener* ***ningún tabulador*** *ya que los manifiestos no pueden contenerlos.*

A.2.4. Unión de nodos al *cluster*

Una vez el maestro ha sido configurado correctamente y se ha desplegado una red de *pods*, se puede proceder a instalar los paquetes necesarios en el resto de nodos. (Estos paquetes se indican en la Subsección A.2.1).

Tras la instalación de los paquetes, se procede a la unión desde los nodos al maestro. Para ello se entra como superusuario y se ejecuta

```
> kubeadm join --token [TOKEN] [MASTER-IP]
```

indicando el token que **kubeadm init** proporcionó anteriormente en el maestro y la IP del maestro.

Para comprobar que correctamente se han unido los nodos al *cluster*, se ejecuta **kubectl get nodes** en el maestro y se obtiene información de los nodos del *cluster*, como nombre del *host* e información relativa a si está preparado y sin problemas (*ready*) o tiene algún error. También se puede obtener información sobre un nodo concreto con *kubectl describe nodes [nombre-nodo]* que informa sobre IDs, logs, recursos o *pods* desplegados.

A.2.5. Reseteo de la configuración kubeadm

Para concluir con esta sección se van a indicar algunos aspectos para hacer un reinicio de la configuración del *cluster*. El comando **kubeadm init** realiza una serie de modificaciones y creaciones de ficheros de configuración que en ocasiones puede ser

necesario eliminar para volver a crear un *cluster* o eliminar una serie de nodos del mismo. Hay varios comandos principales para el reseteo de la configuración de los nodos, *todos ellos necesitan de ser superusuario para ejecutarlos*:

- **kubeadm drain**: Este comando hará que el nodo en el que se ha ejecutado sea marcado para no poder ser utilizado para desplegar *pods*, es decir, no se va a utilizar en el *cluster*. Es muy útil para realizar mantenimiento o para hacer un desunión del nodo de forma segura y correcta. En este comando los *flags* también son interesantes e importantes antes de usarlo:
  - ***–delete-local-data***: Continuará con el comando incluso si hay *pods* utilizando datos locales. Estos, además, serán eliminados durante el drain.
  - ***–force***: Se fuerza en el caso de que haya *pods* que no estén en el conjunto de Réplicas o en un StatefulSet.
  - ***–grace-period int***: Con este *flag* se puede indicar el periodo de tiempo en segundos que se da a los *pods* para acabar de forma correcta y segura. Por defecto es -1 y si se indica un valor negativo se usará -1.
  - ***–timeout duration***: Tiempo hasta que se vuelve a marcar el nodo como activo *("scheduled")*. Es muy útil para realizar tareas de mantenimiento.
- **kubeadm uncordon**: Este comando es el cual vuelve a marcar el nodo como activo, es decir, permite que se vuelvan a poder desplegar *pods* en el nodo. Se usa para volver a activar el nodo tras un periodo de *"drain"*.
- **kubeadm reset**: Comando muy importante cuando se quiere realizar un reseteo de la configuración del *cluster*. Este comando elimina los ficheros que se crearon con *kubeadm init* y permite volver a una configuración inicial. No se debe ejecutar dos veces seguidas el *kubeadm init*, esto provocará el fallo del segundo comando y no realizará un reseteo. Para lograrlo se debe hacer *kubeadm reset* y posteriormente *kubeadm init*. Hay que estar muy atento a la salida de este comando ya que en ocasiones indica que no se ha podido eliminar ciertos ficheros por lo que habría que eliminar a mano.

A.3. Debugging

Obtener información del *cluster* es un paso necesario tras realizar cualquier cambio en el mismo. En todo momento, se puede necesitar conocer la razón de fallo del sistema, conocer sobre que nodos se están ejecutando los distintos *pods* o si es posible que falten recursos.

A.3.1. Comandos más frecuentes

Casi todos los comandos relativos a kubernetes, y herramientas que utiliza, tienen posibilidad de obtener información sobre ellos. Se van a indicar qué comandos existen, qué información pueden proporcionar y para qué se puede utilizar dichar información.

Kubectl logs

El comando kubectl tiene una gran cantidad de comandos para obtener información. El primero que se nombra es ***kubectl logs***, el cual permite estudiar los logs de los *pods*. Para ello simplemente se debe indicar el nombre del *pod* que se quiere estudiar y si se desea una serie de *flags*.

```
> kubectl logs [-f] [-p] POD [-c CONTAINER] [options]
```

Con [-f] se indica si se quiere transmitir el log y con [-p] se pide información de una instancia anterior del *pod*.

El resto de opciones permite, por ejemplo, indicar información sobre la fecha de inicio de los logs (–timestamps); pedir los últimos logs (–tail=int), por defecto se muestran todos; pedir los logs más recientes de un duración relativa (–since=int), como 5s, 4m o 3h, por defecto se muestran todos también.

Kubectl get

Es, posiblemente, uno de los comandos más completos y con más posibilidades para debuggear el *cluster*. El comando genérico es:

```
> kubectl get (TYPE NAME) [flags]
```

Kubectl get nodes

Permite obtener una visión general sobre los nodos del *cluster*. Incluye información sobre el nombre del nodo, el estado del nodo y el tiempo que lleva activo.

El valor de [status] puede variar:

- *Ready*: el nodo se está ejecutando correctamente.
- *OutOfDisk*: ya no tiene recursos para desplegar más *pods* pero se están ejecutando correctamente los que tiene.
- *NotReady*: hay algún problemas con el nodo.
- *NotScheduled*: se ha indicado al *scheduler* que no utilice este nodo para desplegar *pods*.

Esta información es bastante limitada por lo que se recomienda utilizar otros comandos que proporcionan más información.

Kubectl get *pods*

La base del *cluster* Kubernetes son los *pods*. Estos permiten ejecutar las aplicaciones de forma rápida y sencilla y por tanto es necesario llevar un control de su estado, localización, *namespace*, etc. Esta es la forma de ejecutar *kubectl get pods* en los casos generales.

```
> kubectl get pods [-n NAMESPACE] -o wide
```

Se debe incluir el *namespace* sobre el que se quiere buscar la información de *pods*. La primera posibilidad es mostrar todos los *pods* del sistema (*en vez de -n namespace, –all-namespaces*) lo cual permite ver el namespace de cada *pod* y su información. Si no se indica el namespace se mostrarán los *pods* del *namespace=default*. La opción de -o wide se recomienda utilizar siempre, ya que muestra más información necesaria para controlar el *cluster*.

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
default	nginx0-2564987040-sb743	1/1	Running	1	2d	10.36.0.19	k3
default	nginx1-2734069922-wr2t2	1/1	Running	1	2d	10.44.0.6	k2

Figura A.2: Ejemplo de salida de kubectl get *pods* -o wide

Se mostrará el namespace, nombre del *pod*, su estado, si ha tenido que resetearse, tiempo que tiene, IP y nodo en el que está desplegado. Existen diferentes estados que nos indican la situación actual de los *pods*:

- *Pending*: El *pod* ha sido aceptado por el sistema de Kubernetes y está esperando la creación de una o más imágenes de contenedores. Este estado incluye el tiempo hasta que ha sido asignado a un nodo así como el tiempo de descarga de las imágenes. Este estado también puede deberse a que el *scheduler* no es capaz de desplegar el *pod* en ningún nodo, por ejemplo, por falta de recursos.

- *Running*: El *pod* ya ha sido asignado a un nodo y todos los contenedores han sido creados de forma correcta.
- *Succeeded*: Todos los contenedores del *pod* han terminador de forma satisfactoria y no se van a reiniciar.
- *Failed*: Al menos uno de los contenedores del *pod* han acabado de forma incorrecta.
- *Unknown*: Indica que por alguna razón el estado del *pod* no puede ser obtenido, normalmente es debido a un error de comunicación entre el *host* y el *pod*.
- *CrashLoopBackOff*: Este estado indica que se ha creado el *pod* pero se queda bloqueado durante el reinicio del mismo. Puede ser debido a una mala formación del manifiesto de creación. El sistema kubelet (sistema encargado de supervisar el correcto funcionamiento de los *pods*) detecta que está reiniciando un *pod* que acaba constantemente, tras varias veces de bucle pasa al estado CrashLoopBackOff.

kubectl get services

Un servicio de Kubernetes es una abstracción que define un conjunto de *pods* de forma lógica y una política para acceder a ellos. El primer ejemplo de servicio al trabajar con Kubernetes es el dashboard.
Este comando permite ver cuales son los servicios del *cluster* además de cierta información sobre estos. Este comando tiene una salida muy similar a los anteriores (ya que pertenece a kubectl get) por lo que funciona de forma similar.

```
> kubectl get services [-n NAMESPACE] -o wide
```

kubectl get events

Comando muy importante para saber el estado general de los distintos eventos del sistema. Al igual que la mayoria de comandos de kubectl, se debe indicar el namespace.

```
> kubectl get events [-n NAMESPACE]
```

Con este comando se ve todo lo que ocurre en el *cluster*, cuando ha ocurrido, donde ha ocurrido y mensajes de log que el propio sistema manda. Se recomienda utilizar este comando en el caso de no saber concretamente donde esta el error que haya surgido o si se quiere tener información de los nodos y *pods* del *cluster*.

kubectl describe

Una vez localizado el problema (*namespace*, *pod*, servicio o nodo en el que ha ocurrido) se debe conocer cual ha sido la razón del error y de esta forma buscar una solución. Se puede describir cualquier parte del sistema y cada una de ellas tiene una salida con información relativa a ese componente.

Para conseguirlo, se debe utilizar este comando, siempre indicando el *namespace* (si no se indica se toma el por defecto y puede parecer que el recurso no exista).

```
> kubectl describe (-f FILENAME | TYPE [NAME_PREFIX | -l label] | TYPE/NAME)
```

Este comando no se debe utilizar solamente en caso de error, también se debe y se puede utilizar para obtener información para realizar accesos a servicios por ejemplo. A continuación, se comentan los más importantes y qué información se obtienen de cada uno de ellos.

kubectl describe *pods*

Este comando permite al usuario comprobar cual es el estado de un *pod* además de proporcionar gran cantidad de información relativa a la creación del *pod* como puertos que utiliza, la imagen del contenedor o el estado. La última sección del comando nos indica los eventos que tiene el *pod*, es decir, explica cual es el estado y los pasos que ha recorrido. En caso de que todo haya ido bien se verán los pasos de su creación como *Scheduled* (asignado a uno de los nodos), *Pulled* (en el momento que la imagen del contenedor esté en la máquina, se descargará si no lo está), *Created* (el contenedor ya ha sido creado) o *Started* (en el momento que el *pod* ya esté funcionando).

En cambio, si ha ocurrido algún error se *pod*rá ver con este comando también. Los eventos que se muestren dependeran del error que hayan llevado al *pod* a no funcionar correctamente. Por ejemplo, si se queda en *Pending* puede ser por que los recursos que solicita el *pod* son superiores a los recursos de los nodos, en este caso mostrará *FailedScheduling* indicando la razón (si se supera el valor de CPU requerido se indicará *Insufficient cpu*). Para solucionarlo se pueden eliminar los *pods* que ya no se usen, añadir nuevos nodos al *cluster* o, incluso, modificar el manifiesto y limitar los recursos lo que pueden estar provocando un malfuncionamiento de la aplicación.

Otro error que puede ocurrir es que no se encuentre la imagen del contenedor, en este caso al hacer **kubectl get *pods* -n namespace** se verá el error *ErrImagePull* y con **kubectl describe *pods* nombrePod -n namespace** se podrá ver que no ha encontrado

la imagen (*FailedSync*) en el respositorio de docker.io que es donde se almacenan las imágenes que son descargadas.

kubectl describe nodes

Este comando es muy interesante para conocer los recursos que quedan disponibles y cuales están en uso actualmente sobre cada uno de los nodos (uso de CPU, memoria o *pods* ejecutándose en cada uno de ellos). Se recomienda utilizarlo en el caso de que algún *pod* quede en estado de *Pending* y no haya podido ser asignado a ningún nodo, para conocer cuántos recursos quedan en cada nodo y la razón por la que no se ha desplegado.

kubectl describe services

Como su propio nombre indica, este comando va a ser utilizado para obtener información sobre los servicios del *cluster*. Por ejemplo, se utilizará en caso de querer conocer un puerto de salida de un servicio para poder acceder a él. Se obtendrá información similar al manifiesto que creó el servicio, como puertos de funcionamiento, *namespace* al que pertenece, etc.

A.3.2. Dashboard

Además de los comandos propios de Kubernetes como es kubectl, existe una serie de manifiestos que posibilitan comprobar el estado general del sistema a través de una interfaz gráfica, como es el Dashboard [1]. Este sencillo manifiesto crea un servicio al cual se accede a través del navegador web y en el cual se puede obtener gran información del *cluster*. Se puede observar los recursos que utilizan los *pods*, los servicios que el *cluster* tiene, los nodos con toda su información, etc.

Se recomienda desplegar este manifiesto que crea un *pod* y un servicio ya que permite ver el sistema de una forma muy sencilla y rápida, incluso permitiendo realizar cambios en el mismo desde la propia interfaz. Todos los estados de los que se ha estado hablando en esta sección se pueden comprobar desde la interfaz y ver gráficas de uso de recursos.

A.3.3. Conclusiones

Sin duda, cada uno de estos comandos deben ser conocidos por todo aquel que quiera trabajar con el sistema Kubernetes ya que proporcionan la cantidad necesaria de

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

información para debuggear el sistema, conocer los errores, donde ocurren, el porque de estos y lo que es más importante, gracias a esta información se podrán solucionar de una forma sencilla.

Para terminar, hay que tener claro que se debe complementar el uso de estos comandos y el dashboard, es decir, siempre se debe utilizar dos o más de estos comandos para conocer el verdadero motivo de los eventos que ocurren en el sistema y tener claro cual es la información que se puede obtener de ellos ayudará a ser más ágil en el proceso de debugging del sistema. .

Anexo B

Flink

B.1. Introducción

Flink es un *framework open-source*, escrito en JAVA, para el procesamiento de datos en *streaming*, aunque también es posible trabajar por lotes. Según los desarrolladores, proporciona resultados precisos, incluso en caso de la llegada de datos fuera de tiempo; tiene estado y es tolerante a fallos, recuperándose de errores mientras mantiene un estado de la aplicación y funciona a gran escala, ejecutándose en miles de nodos.

Flink permite el procesado en *data stream* y en ventana con eventos de tiempo, es decir, cada cierto tiempo de los datos en *streaming* se obtiene unos resultados. De esta manera es más fácil obtener resultados más precisos para los eventos que vengan desordenados o tarde.

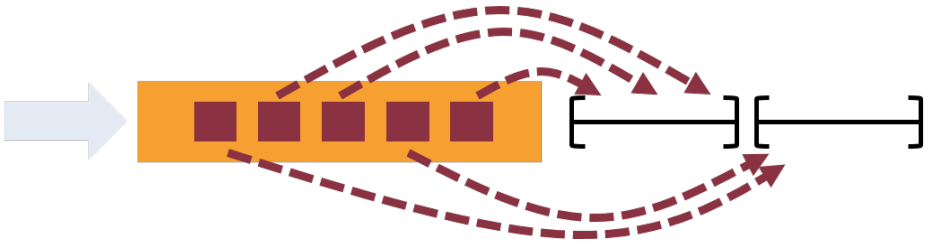


Figura B.1: Ilustación del procesado de datos de Flink. [1]

Además estas ventanas pueden ser personalizadas según el tiempo o número de datos para soportar patrones de procesado más complejos. De esta forma, se adapta a los modelos de realidad de los datos.

[1] Imagen obtenida de: `https://flink.apache.org/introduction.html`

Además permite realizar un procesamiento por lotes, batch, si los datos que se han obtenido no llegan en flujo sino que son finitos.

B.2. Operativa de Flink

El *framework* Flink tiene un funcionamiento similar a otros sistemas de Maestro-Esclavo. En este caso, el maestro es el Job-Manager y los esclavos son los Task-Managers, sobre los cuales se realizan las operaciones. El Job-Manager es el encargado de recibir los trabajos a realizar y decidir sobre que Task-Manager se debe ejecutar dicho trabajo.

Cada uno de los Task-Manager tiene un número de task-slots que son los espacios de trabajo. Estos corresponden con cada uno de los procesadores, es decir, una máquina con 2 procesadores hará que cada Task-Manager tenga 2 task-slots. Esto indica que si hay varios Task-Managers sobre una misma máquina van a compartir los task-slots aunque el Job-Manager va a suponer que tiene la suma de todos, es decir, que van a compartir los procesadores a lo hora de trabajar.

Además del funcionamiento básico de Flink, hay que indicar que tiene una serie de configuraciones personalizables que permiten modificarlo. La que se considera muy importante es el paralelismo *parallelism.default*, esto es, sobre cuántos task-slots se realiza cierta operación. Por defecto, es uno, por lo que si se manda un trabajo al Job-Manager que tiene un Task-Manager con 2 task-slots solamente se va a utilizar uno de ellos, es decir, un único procesador. Hay que precisar que Flink permite el paralelismo en todos sus trabajos pero si se quiere modificar se debe hacer de forma explícita al mandar el trabajo al Job-Manager. Toda la configuración personalizable de Flink puede encontrarse en [2].

Otra configuración importante es el número máximo de mensajes que se permiten enviar entre el Job-Manager y los Task-Managers (*akka.framesize*). En caso de que un trabajo trate de enviar más mensajes (en bits) que los indicados en la configuración, este fallará. Si se ha detectado este posible error, se debe modificar la configuración aumentado este valor.

B.3. Flink sobre Kubernetes

Una vez comprendidos los diferentes conceptos relativos a Kubernetes y a Flink, se tiene como objetivo juntar ambas tecnologías para que Flink se apoye en Kubernetes. Esto es, crear una serie de *pods* y servicios que permitan ejecutar de forma automática las aplicaciones de procesamiento de datos de Flink.

[2] `https://ci.apache.org/projects/flink/flink-docs-release-1.3/setup/config.html`

Para realizar esto se ha investigado y se ha encontrado una serie de manifiestos [3] para Kubernetes que ponen en marcha un Job-Manager y uno (o varios) Task-Managers de Flink. Se compone de 5 manifiestos:

- *namespace.yaml*: Creación del espacio de nombres sobre el que estarán los *pods* y servicios de la aplicación.
- *jobmanager-controller.yaml*: Creación del Flink Job Manager.
- *jobmanager-service.yaml*: Creación de un servicio lógico que funciona como endpoint para que los Task Manager puedan acceder al Job Manager.
- *jobmanager-webui-service.yaml*: Creación del servicio de interfaz web que dará información sobre la aplicación.
- *taskmanager-controller.yaml*: Creación del Task Manager, se puede indicar el número de réplicas que se quieran.

NOTA: Hasta que no se haya comprobado que el pod *del jobmanager-controller este ejecutandose de forma correcta (Running) no se debe desplegar los Task Managers.*

Estos manifiestos permiten desplegar un sistema con una configuración que viene por defecto en la imagen de Docker. Esto ha supuesto un problema ya que para realizar ciertos experimentos era necesario modificar esta configuración. Para el caso del paralelismo se ha solucionado de forma sencilla ya que al mandar un trabajo al Job-Manager es posible indicarle el paralelismo a través de un parámetro. En cambio, en otro de los experimentos (`PageRank`) el número de mensajes enviados entre Job y Task-Manager era de una cantidad mayor que la definida por lo que se ha tenido que realizar una copia de la imagen de Docker y crear una nueva modificando esta configuración.

[3] `https://github.com/melentye/flink-kubernetes`

Anexo C

Thrill

C.1. Introducción

Thrill es un *framework* en C++ para procesamiento de Big Data de forma distribuida en un *cluster* de máquinas. Se encuentra en un estado temprano de desarrollo y pruebas. Thrill trata de dar respuesta a los mismos problemas que Flink pero buscando un mejor rendimiento a través de un lenguaje como C++.

C.2. Operativa de Thrill

Thrill tiene dos formas de ejecución posibles, en una máquina o en un *cluster*. Para la primera basta con tener un compilador apropiado a la versión C++14 (versión en la que está escrito Thrill), clonar el repositorio de Thrill [1] y compilar. La compilación clonará repositorios necesarios, creará el subdirectorio *build*, utilizará *cmake* para crear Makefiles y compilará en modo *debug* para finalmente realizar una serie de test (actualmente 95).

Para el modo de *cluster* es necesario un protocolo a nivel de red que comunique cada uno de los *hosts*, como NFS, o que los ficheros estén presentes en la máquina.

Si se ejecuta localmente, como se ha explicado antes, se simula un sistema con tantos *hosts* como núcleos tenga la máquina y haciendo la comunicación mediante los *sockets* del *kernel* de Linux.

En cambio, para la ejecución de Thrill en un *cluster* se van a utilizar tantos *hosts* como máquinas tiene el mismo con tantos trabajadores como núcleos tiene la máquina.

[1] `https://github.com/thrill/thrill.git`

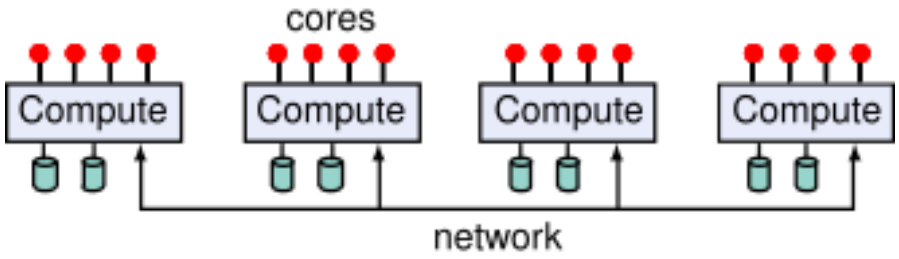


Figura C.1: Ejemplo de *cluster* para ejecutar Thrill. [2]

Esta ejecución se hace a través de uno de los *scripts* que Thrill proporciona como base de su proyecto, `invoke.sh`. En el mismo, se pueden indicar los *host* sobre los cuales se va a ejecutar el trabajo indicado. Este *script* se encarga de copiar el ejecutable sobre la máquina de trabajo, pero necesita que la máquina destino tenga los ficheros de entrada presentes. Finalmente, se ejecuta sobre cada *host* utilizando todos los núcleos que tenga la máquina.

NOTA: Para cada una de las ejecuciones Thrill proporciona diferentes scripts en su código fuente. Tras la compilación ya se pueden ejecutar aunque para la segunda de ellas es necesario tener una configuración concreta.

C.3. Thrill sobre Kubernetes

A diferencia de Flink, no se ha podido encontrar ninguna imagen con Thrill desplegado, por lo que ha sido necesario desarrollar una con lo necesario para realizar los experimentos de este proyecto. Esta imagen necesita de acceso directo, a través de ssh, a los *hosts* sobre los que se ejecutarán los trabajos de los experimentos.

El código fuente del Dockerfile utilizado para la creación de un contenedor con Thrill se muestra en el Código C.1. Este contenedor parte de la última versión de Ubuntu, instala los paquetes necesarios para la compilación de Thrill, coloca la clave para poder realizar los accesos a las máquinas y, finalmente, compila los ficheros necesarios para la ejecución de Thrill.

La clave de las máquinas es generada e introducida en las máquinas donde se va a realizar la ejecución en el fichero *authorized_keys*. De esta forma los contenedores pueden realizar el acceso a través de ssh sin contraseña para la ejecución.

En el Código C.2 se muestra un manifiesto de ejemplo, para la ejecución de Thrill. En el mismo, se utiliza la imagen de Docker que se ha comentado y se realiza una ejecución a

[2]Imagen obtenida de: `http://i10login.iti.kit.edu/thrill-doxygen/start_run.html`

través de *invoke.sh* sobre un nodo (2dos) de la aplicación `WordCount` y pasando el fichero de prueba como parámetro.

```
# Take last Ubuntu version
FROM ubuntu:latest

RUN apt-get update
RUN p=$(uname -r)
RUN apt-get -y install linux-tools-${p}
# Packages used by Thrill instalation
RUN apt-get -y install build-essential git cmake openssh-client sshpass
RUN mkdir /thrill
ADD * /thrill/
WORKDIR "/thrill"
RUN mkdir /root/.ssh
# This rsa key is given in the files directory
RUN mv id_rsa* /root/.ssh/
#Compile Thrill
RUN ./compile.sh


ENV THRILL_HOME /thrill
ENV PATH $PATH:$THRILL_HOME
```

Código Fuente C.1: Dockerfile Thrill

```
apiVersion: v1
kind: Pod
metadata:
    name: thrill-1000millones
spec:
    restartPolicy: Never
    containers:
        - name: thrill
          image: carlos:thrill
          command: ["/bin/sh", "-c"]
          args: ["echo 'Starting Word Count example in cluster' &&
          cd /thrill/build/examples/word_count/ &&
          /thrill/run/ssh/invoke.sh -c -h \"2dos\" -u ubuntu word_count_run
           /tmp/benchmark/1millon.txt"]
```

Código Fuente C.2: Manifiesto para la ejecución de Thrill sobre Kubernetes.

Anexo D

Código fuente de *scheduler* propuesto

En este anexo se presenta el código fuente del *scheduler* de cliente que se ha presentado como solución en el Capítulo 5. En el Código D.1 se presentan, en primer lugar, todas las funciones auxiliares para la decisión del nodo destino de la aplicación; mientras que en segundo lugar se pueden ver los pasos necesarios a través de las llamadas a las funciones. Al inicio del código, se pueden ver las variables globales como son el fichero de estado del *cluster*, la lista de nodos del *cluster* (una máquina en cada línea) y la tabla de penalización.

```
import subprocess
import sys
from optparse import OptionParser

#-------------------Program Variables---------------#
filename = 'podsStatus.txt'
machineFile = 'maquinas.txt'
penalty = [ [5,4,2,1], [4,3,1,0], [2,1,5,4], [1,0,4,3] ]
#-------------------Cluster Functions----------------#
#Return all the \emph{pods} running in the cluster
def getStatus():
        process = subprocess.Popen(['kubectl', 'get', 'pods',
                        '--all-namespaces', '-o', 'wide'], stdout=subprocess.PIPE)
        out, err = process.communicate()
        return out

#-------------------Aux Functions--------------------#
#Create a list from the line and return the Pod's info
```

```
def getPodInfo(line):
        wordList = line.split()
        result = []
        if len(wordList) > 0 and wordList[0] != 'kube-system' and
                (wordList[3] == 'Running' or wordList[3] == 'Pending'
                 or wordList[3] == 'ContainerCreating'):
                        result.append(wordList[1])
                        result.append(wordList[7])
        return result

def getMinApps(userApps):
        minApps = -1
        for i in range(0, len(userApps)):
                if len(userApps[i]) < minApps or minApps == -1:
                        minApps = len(userApps[i])
        return minApps

def translateLabel(label):
        if label == 'cpu:alto':
                return 0
        if label == 'cpu:bajo':
                return 1
        if label == 'disk:alto':
                return 2
        if label == 'disk:bajo':
                return 3

def calculatePenalty(nodeApp, newApp):
        total = 0
        for i in range(0, len(nodeApp)):
                appPos = translateLabel(nodeApp[0])
                total = total + penalty[newApp][appPos]
        return total

#Given a userLabel <<labels>> decide the dest
#node depending on running apps <<userApps>>
def decideMachine(labels, userApps, nodes, minApps):
        if len(userApps) == 0:
                return nodes[1]
        else:
                for i in range(0, len(userApps)):
                        if len(userApps[i]) == 0:
                                return nodes[i]
                newApp = translateLabel(labels[0])
```

```
                destMachine = ''
                penaltyValue = 1000
                for i in range(0, len(userApps)):
                        if minApps == len(userApps[i]):
                                penaltyNode = calculatePenalty(userApps[i], newApp)
                                if penaltyValue > penaltyNode:
                                        penaltyValue = penaltyNode
                                        destMachine = nodes[i]
                return destMachine

#-----------------File functions--------------------#
#Delete a line of a \emph{pod} that is not running alredy
def deleteLine(filename, name):
        f = open(filename,"r")
        lines = f.readlines()
        f.close()
        f = open(filename,"w")
        for line in lines:
                if name not in line:
                        f.write(line)

#Get user labels from the manifest <<name>>
def getUserLabels(name):
        f = open(name, 'r')
        line = f.readline()
        if 'userLabels' not in line:
                print 'File malformed, the first line must be: userLabels;
                cpu:<<alto|bajo>>;mem:<<alto|bajo>>'
                return []
        labels = line.split(';')
        userLabels = []
        userLabels.append(labels[1].rstrip())
        return userLabels

#Save the new app in the status file
def saveUserApp(name, node, userLabels):
        f = open(filename, 'a')
        f.write(name + ' ')
        f.write(node + ' ' )
        f.write(userLabels + '\n')
        f.close()

#From a manifest get the name of the App
def getAppName(name):
```

```
        f = open(name, 'r')
        lines = f.readlines()
        for line in lines:
                if 'name' in line:
                        return line.split(':')[1].strip()

#Given a list of running \emph{pods} <<podsList>>, delete Apps
                                #that were in the Status File but are not running.
def deleteApps(podsList, name):
        f = open(name, 'r')
        lines = f.readlines()
        f.close()
        f = open(name, 'w')
        for line in lines:
                lineList = line.split(' ')
                for pod in podsList:
                        if lineList[0] in pod[0] and lineList[1] == pod[1]:
                                f.write(line)

#Replace in the manifest the nodeSelector key to assign
                                #the pod to the machine
def addNodeSelector(manifest, machine):
        f = open(manifest, 'r')
        lines = f.readlines()
        f = open(manifest, 'w')
        for line in lines:
                if 'machine' in line:
                        line = line.replace('machine', machine)
                f.write(line)
        f.close()

#From the Status File it returns a list with pairs: <<node, label>>
def getFileApps(nodes):
        f = open(filename, 'r')
        userApps = []
        for node in nodes:
                userApps.append([])
        for line in f.readlines():
                line = line.split(' ')
                for i in range(0, len(nodes)):
                        runningNode = line[1].strip()
                        if nodes[i] ==  runningNode:
                                userApps[i].append(line[2].rstrip())
        return userApps
```

```
#From the machine File get nodes of the cluster
def getNodes():
        f = open(machineFile, 'r')
        nodes = []
        for line in f.readlines():
                nodes.append(line.rstrip())
        return nodes


################# MAIN #################
parser = OptionParser()
parser.add_option("-n", "--namespace", dest="namespace",
        metavar="namespace.yaml", help="Manifest of the namespace")
parser.add_option("-a", "--app", dest="app", metavar="infraestructure file",
                  help="File with all manifests that this app needs,
                  will be creating in the given order.")
(options, args) = parser.parse_args()

if len(args) == 0:
        print "At least one manifest required"
        print "Usage: \n\tpython solution.py [-n namespace] <<Manifests>>"
        sys.exit()
#manifest = 'thrill-1millon.yaml'
manifest = args[0]

#1. Get current cluster Status
salida = getStatus()
podsList = []

#2. Get \emph{pods} running in a list
for line in salida.split('\n'):
        aux = getPodInfo(line)
        podsList.append(aux) if len(aux) > 0 else ''

#3. Delete \emph{pods} that was in the file but are not running any more.
deleteApps(podsList, filename)

#4. Get user labels
labels = getUserLabels(manifest)
if len(labels) == 0:
        sys.exit()

#5. Create the correct manifest
```

```
appName = getAppName(manifest)
deleteLine(manifest, 'userLabels')

#6. Decide the dest machine
nodes = getNodes()
userApps = getFileApps(nodes)
minApp = getMinApps(userApps)
machine = decideMachine(labels, userApps, nodes, minApp)

#7. Add the machine label to the manifest
addNodeSelector(manifest, machine)

app = options.app
if app != None and len(app) > 0:
        l = getInfMan(app)
        for man in l:
                addNodeSelector(man, machine)
                subprocess.Popen(['kubectl', 'create', '-f', man], stdout=subprocess.PIPE)

#8. Save the new App in the status File.
userLabels = labels[0]
saveUserApp(appName, machine, userLabels)

#9. Create the pod.
process = subprocess.Popen(['kubectl', 'create', '-f', manifest], stdout=subprocess.PIPE)
out, err = process.communicate()
```

Código Fuente D.1: Código del *scheduler* presentado como solución.