Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza

Proyecto de fin de Carrera
Ingeniería en Informática
Curso 2017/2018

# Estudio de políticas de despliegue y *scheduling* en una federación de Kubernetes

**Guillermo Robles Gonzalez**

Director: Víctor Medel Gracia

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

2018-06-29

# Diseño de políticas de scheduling para una federación de Kubernetes


## RESUMEN

La abstracción de federación ofrece la posibilidad de compartir recursos computacionales entre elementos heterogéneos de manera transparente. Paradigmas cómo el *Edge Computing* pueden beneficiarse de estos planteamientos, permitiendo satisfacer determinados requisitos no funcionales (cómo menor coste, facilidad de gestión, disminución de latencias de red…). En el presente trabajo se plantea una arquitectura basada en un sistema de gestión de contenedores (Kubernetes) para aprovechar las ventajas que este sistema pueden ofrecer en un entorno de federación.

Se han analizado las posibilidades que ofrece estos modelos en un caso de uso particular, como es el análisis de imágenes, lo cual permite observar el comportamiento del sistema bajo una carga realista. El sistema diseñado e implementado se basa en aprovechar la capacidad computacional del *Edge*, de coste más bajo y latencia menor, en vez de procesar todos los elementos en el *Cloud*. Además, se ha utilizado la abstracción de federación para gestionar y compartir recursos entre diferentes *Edges* de manera transparente. Durante el desarrollo del trabajo ha primado un análisis basado en evidencias empíricas, apoyando cualquier decisión de diseño importante en datos obtenidos de manera experimental.

Para la realización de los experimentos sea ha realizado la implementación en un sistema real heterogéneo, realizando un despliegue en varios clústers físicos, formados tanto por máquinas *x86_64* clásicas como *ARM* de menor coste y rendimiento (Raspberry Pi), a fin de explorar de forma más detallada el comportamiento del sistema diseñado en distintas situaciones. Durante los experimentos, se ha conseguido una mejora notable, llegando a alcanzar un factor de reducción de coste económico de casi 6, con un bajo efecto sobre el rendimiento del sistema. Se puede ver que estos sistemas tienen amplio margen de mejora, estableciéndose cómo potenciales competidores a las infraestructuras basadas en *Cloud* usadas en la actualidad para determinados ámbitos aplicativos.



**Palabras clave**: kubernetes, federation, scheduling, edge computing

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. Guillermo Robles Gonzalez,

con nº de DNI 73131695-k 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, (Título del Trabajo) Diseño de políticas de scheduling para una federación de Kubernetes,

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

Zaragoza, a 29 de Junio de 2018

Fdo: Guillermo Robles

# Índice general

# Índice de figuras

# Capítulo 1

# Introducción

En los última década, el paradigma de computación *Cloud* se ha extendido rápidamente, cambiando la forma de diseñar, implementar y gestionar sistemas distribuidos. Esta filosofía consiste en mover la potencia computacional a la "nube" (*Cloud*, en términos anglosajones). Esta abstracción es un sistema, normalmente externo y potencialmente alejado geográficamente, que ofrece gran cantidad de recursos de computación o almacenaje, de manera escalable, transparente, y accesible. Estos recursos, sin embargo, suelen tener un coste – económico o de prestaciones en acceso – que puede supera el de los recursos locales. Este paradigma se ayuda de métodos cómo la virtualización para lograr el aislamiento entre entornos o clientes diferentes[1]. Esto implica una complejidad adicional asociada a la gestión del conjunto de recursos virtualizados, entre los cuales se puede incluir red, almacenamiento, o computación.

De esta primera aproximación, y cómo contrapunto o alternativa, aparecen paradigmas derivados cómo el *Edge Computing*[2][3] o el *Fog Computing*[4]. Si el *Cloud*, que desde el punto de vista de un cliente se puede ver cómo un modelo de caja negra[5] en el que existe un sistema cerrado, alejado de las máquinas que consumen su servicio; el *Edge* y el *Fog* defienden un modelo distribuido, en el cual se pueden aprovechar la cada vez mayor potencia de los elementos existentes cercanos a los sensores y clientes para hacer trabajos. Las ventajas del *Edge* con respecto al *Cloud* son amplias: incremento de la seguridad (dado que la información nunca sale de la red interna); incremento de la disponibilidad (dado que no es necesaria la conexión a Internet, lo cual puede ser beneficioso en comunidades rurales o en vías de desarrollo) o incremento del aprovechamiento (dado que permite explotar al máximo posible los recursos existentes)[6][7][8].

Las federaciones son una forma de conectar sistemas heterogéneos, como los que pueden surgir en el paradigma de *Edge Computing*, de forma que se pueda tener un sistema centralizado de gestión o monitorización[9]. Este sistema centralizado nos ofrece una abstracción frente a las complejidades de estos sistemas independientes,

y nos permite trabajar con todos los recursos combinados de los participantes de manera transparente. Aunque la visión más clásica de la federación considera sobre todo las capacidades de gestión transparente y centralizada, este modelo también nos permitiría conectar sistemas diferenciados, y mover información y recursos de unos a otros. Esto permitiría tener un conjunto de participantes en la federación, de forma que cada uno de ellos ofreciera un conjunto de recursos de computación, consumibles por todos los componentes de la federación. De esta forma se crea un conjunto de recursos centralizados, más cercanos (Especialmente si se consideran federaciones de entidades cercanas geográficamente) y, en principio, más baratos que el *Cloud*. Estas federaciones pueden estar formados por cualquier entidad que pueda ofrecer potencia de computación y tenga necesidad de procesados de los mismos, pudiendo ayudarse mutuamente, sin necesidad de ir directamente a las opciones ofrecidas por el *Cloud*.

Todo esto se sustente en el uso de máquinas virtuales (en adelante MVs) cómo sistema de aislamiento. Esto consiste en crear, dentro de servidores o máquinas potentes, zonas aisladas mediante la virtualización de hardware, que nos permitan gestionar lo que antes serían gran cantidad de máquinas físicas cómo dispositivos virtuales y ficheros de discos y configuración. Este paradigma tiene evidentes ventajas: desde la posibilidad de destruir y crear máquinas a gran velocidad (comparado con el coste, en tiempo y dinero, de la instalación de los servidores físicos), moverlas entre ubicaciones geográficamente separadas a través de la red o la posibilidad de sobreprovisionar nodos físicos. Sin embargo, este modelo no esta exento de problemas, yendo estos desde el alto consumo de recursos que conlleva la virtualización hasta la gestión de los hipervisores (cómo ESXi, Xen, KVM…), además de las dificultades implícitas de gestionar una granja de servidores físicos. A partir de estos modelos se desarrolla el concepto de los "contenedores de aplicación", similares en funcionalidad a las máquinas virtuales, y que intentan resolver algunos de los problemas que estas conllevan. Algunos de estos problemas serían tamaño ocupado, reparto de recursos, o velocidad de creación. En definitiva, suponen un método más ligero y flexible de compartir recursos

La relativa juventud de los sistemas de contenedores hace que surjan nuevos retos[10]. Su novedad provoca que no existan herramientas ni protocolos extendidos en la industria para su gestión, haciendo esto que muchas empresas tengan problemas para adaptarse al cambio de filosofía que conlleva el movimiento de máquinas virtuales a contenedores, con el cambio de visión y filosofía que esto lleva. Cómo respuesta a estas carencias surgen proyectos como Kubernetes (a veces denominado "k8s"), el cuál persigue crear una infraestructura completa basada en contenedores, aprovechando las ventajas que estos ofrecen, además de otras características no funcionales, cómo facilidad, escalabilidad, o flexibilidad. Kubernetes permite crear clústers que soporten contenedores encima de máquinas físicas (o

máquinas virtuales, o incluso otros contenedores), de forma que se pueda gestionar infraestructuras de tamaño notable de forma centralizada y sencilla[11][12].

El presente trabajo pretende explorar las posibilidades de estos sistemas, y se estructura en 5 capítulos, siendo el presente la introducción1, en el cuál se incluyen además los objetivos y la motivación que lleva a realizarlo. En el capítulo 2 se incluye el Estado del Arte, centrándose en los aspectos de virtualización y gestión de la misma. También se comentan aspectos particulares de federaciones y *Edge Computing*, incluyendo las particularidades de su *scheduling*. En el capítulo  se concreta el problema a tratar y cómo se ha desarrollado, explicando el caso de uso particular, la arquitectura y los detalles de implementación particulares. Posteriormente, en el capítulo 4 se comentan las particularidades de los experimentos realizados, centrándose en el entorno experimenta, los experimentos preliminares realizados y sus implicaciones, la metodología experimental seguida y los resultados obtenidos y sus implicaciones. A continuación, se presenta una conclusión, incluyendo esta potenciales vías de desarrollo posterior. Finalmente, se presenta un conjunto de Anexos, que pretenden detallar ciertos aspectos que, si bien importantes para el desarrollo del trabajo, no se incluyeron en el cuerpo principal por varios motivos.

## 1.1 Objetivos

Como parte fundamental del Trabajo Fin de Grado, se han establecidos los objetivos que se detallan a continuación:

- **O1– Analizar y diseñar políticas de scheduling de federación**: El uso de la federación implicará que ante diferentes cargas de trabajo de sus miembros, se planteen políticas de reparto de carga entre cada uno de ellos.
- **O2– Desplegar clústers de Kubernetes en diferentes configuraciones, incluyendo esquemas de *Edge Computing***: se pretende utilizar la abstracción de federación para gestionar clústers Kubernetes heterogéneos. De esta manera, cada clúster representará un *Edge* con sus propias peculiaridades, tanto a nivel de infraestructura como de carga de trabajo.
- **O3– Analizar las implicaciones del uso de una federación en Kubernetes**: La federación de Kubernetes es una opción nueva y poco desarrollada, por lo que se analizará tanto el comportamiento y las capacidades de la federación cómo sus potenciales usos para el proyecto a desarrollar.
- **O4– Evaluar experimental de la propuesta de scheduler de federación**: esta evaluación, encuadrada dentro de la metodología científica, se realizará con un entorno real y bajo diferentes parámetros. El objetivo es determinar empíricamente la propuesta obtenida de O1 y O2.
- **O5– Conocer las herramientas tecnológicas y aplicaciones relevantes en el ámbito**: Como parte del proceso de aprendizaje del TFG, y dada

la inexperiencia con algunas de las tecnologías propuestas, se plantea como objetivo el manejar y conocer las diversas herramientas tecnológicas (principalmente, Docker, Kubernetes, y el lenguaje Go)

## 1.2 Motivación

El aumento de la cantidad, complejidad y accesibilidad de los diferentes dispositivos provoca un incremento de la capacidad de cómputo existente. En otras palabras, cada vez existe mayor potencia cercana al cliente final. Un *smartphone* moderno tiene una capacidad, tanto de almacenaje cómo de cómputo, órdenes de magnitud mayor que un *mainframe* de hace años, y la pone al alcance de la mano de cualquier usuario estándar. Esto ha provocado la aparición de paradigmas de trabajo que pretenden aprovechar esta capacidad, cada vez mayor, para realizar trabajo útil. Uno de estos paradigmas de trabajo es el modelo de *Edge Computing*, modelo que defiende la creación de un sistema de computación distribuido y en malla, conectando directamente a sensores y a clientes finales, con la capacidad de compartir recursos de computación de forma transparente y elástica. Este modelo provoca un incremento en la complejidad de gestión, dado que se deben gestionar de manera más o menos homogénea dispositivos heterogéneos. El objetivo de este trabajo es explorar las posibilidades del *Edge Computing* para un caso de uso particular, cómo es el análisis de imagen en tiempo real, proponiendo soluciones a nivel de administración y gestión del sistema, basadas en el concepto de federación.

# Capítulo 2

# Trasfondo Tecnológico y Estado del Arte

En este capítulo se detallará el contexto tecnológico y el Estado del Arte actual, detallándose los aspectos más interesantes de los mismos, de forma que se pueda enmarcar adecuadamente el presente trabajo dentro de las investigaciones actuales de temática similar.

## 2.1 Técnicas de virtualización

Conforme se han popularizado los entornos multi-cliente (inicialmente los mainframes y más actualmente los servidores ofrecidos por una *Cloud* pública), se ha puesto de manifiesto la necesidad de crear técnicas de aislamiento entre las distintas cargas de trabajo que existen dentro de estos sistemas, de forma que los distintos usuarios no puedan perjudicarse entre ellos, bien inconscientemente o bien con objetivos maliciosos..

El concepto de la virtualización cómo método de aislamiento de procesos dentro de sistemas compartidos no es nada nuevo, habiendo existido desde los más antiguos mainframes[13]. Propuestas más sofisticadas surgen de lo sistemas de virtualización basados en el aislamiento de procesos. Ejemplos de estos sistemas son las jaulas (*Jails*) de FreeBSD[14], o las zonas (*Zones*) de Solaris[15]. Con el tiempo, estos sistemas han colapsado en la virtualización a nivel de hardware que hoy en día es el estándar, siendo una herramienta básica en el mundo de la Tecnología de la Información, pervasivo a todos los niveles de la informática[1]. Hoy en día, han resurgido las tecnologías de contenedores (también denominada "virtualiza-

[1]Ejemplo de esta pervasividad son los ISAs de virtualizado, cómo VT-x o AMD-V, o la proliferación de sistemas tipo *Infrastructure as Code*, basados en una plataforma de hardware virtualizado

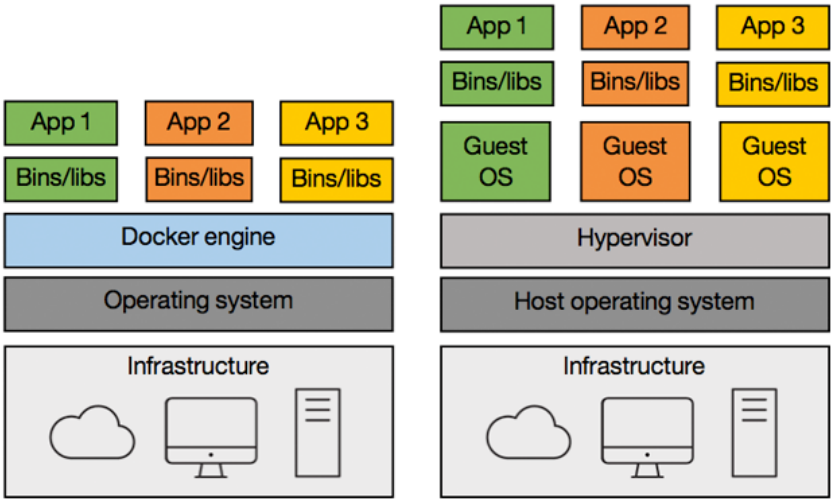



Figura 2.1: Capas de abstracción involucradas en MVs vs. contenedores

ción ligera" o"virtualización de aplicaciones"), las cuales se oponen a las técnicas más clásicas, basadas en la virtualización de sistemas *x86*. En la Figura 2.1 se puede ver las diferentes capas de abstracción involucradas en ambas tecnologías, particularmente Docker cómo tecnología de contenedores.

La diferencia fundamental entre las MV y los contenedores es que mientras las MV se abstraen del hardware del *host*, ofreciendo una interfaz intermedia de hardware configurable y virtual a una MV, la cual ve la máquina tal y como esta capa de abstracción esté configurada. De esta forma se puede restringir recursos como memoria y CPU, cambiar de arquitectura hardware, ofrecer capacidades de aislamiento con otras MV, o incluso crear dispositivos virtuales inexistentes, solo visibles por la MV. Esto hace que una MV tenga un nivel de aislamiento mucho mayor, siendo en algunos casos imposible distinguir si un programa se están ejecutando en una MV, y no en una física[2]. Por el contrario, los contenedores no se abstraen tanto de la máquina física, sino que son simplemente grupos de procesos en una máquina que sólo pueden verse entre ellos[3]. Esto significa que, por ejemplo, los ejecutables en un contenedor han de ser de la misma arquitectura que en el *host*, lo que en entornos heterogéneos añade complejidad de gestión. Pese a ello, los contenedores tienen ventajas importantes con respecto a las MVs, que los hacen más adecuados para gran cantidad de tipos de trabajo. Algunas de estas ventajas

[2]Esto no es siempre cierto. Existen técnicas, como *Blue Pill* (Desarrollado por Joanna Rutkowska) y *exploits* como *Rowhammer* que permiten a un programa en una MV tanto detectar que esta en una MV como afectar a otras MVs en el mismo *host*. Sin embargo, esto son problemas de implementación, no de diseño

[3]Mediante un conjunto de herramientas ofrecidas por el núcleo Linux, entre las que se encuentran *namespace isolation*, *unified hierarchy*, *kernel memory control groups* o *cgroups*

son:

- **Tiempos de arranque**: Dado que un contenedores simplemente tiene que comenzar unos pocos procesos (en comparación con una MV, que ha de arrancar un sistema operativo completo), su tiempo de arranque es notablemente más rápido. Cómo comparativa (asumiendo en ambos casos que ya se ha creado la imagen), mientras que una MV puede arrancarse en alrededor de medio minuto, un contenedor tiene un tiempo de arranque de alrededor de un segundo.

- **Tamaño**: Dado que un contenedor sólo ha de contener la aplicación de interés y sus librerías (aparte de unos pocos ficheros de sistema), su tamaño es notablemente menor. Por ejemplo, para un MV Ubuntu Canonical recomienda un mínimo de 1.5 GB de disco para el sistema[4], mientras que un contenedor Ubuntu ocupa alrededor de 120 MB, una diferencia de más del 90%. Con imágenes especiales (Cómo Linux Alpine, o contenedores ***FROM** scratch*) y compilación estática se puede disponer de un contenedor completo en menos de 10 MB.

- **Facilidad de creación**: Herramientas como *docker build* simplifican el proceso de creación de un contenedor de forma importante, mientras que la instalación y creación de una MV requiere herramientas de cierta complejidad, como Packer o Vagrant.

- **Prestaciones aumentadas**: Dado que los contenedores evitan el gasto extra de emular dispositivos virtuales y el sistema operativo, tienen un consumo de memoria y CPU bastante menores, pudiendo encontrarse mejoras de hasta el 10% en CPU, y aumentos de velocidad de IOPS de hasta el 50%[16].

## 2.2 Gestión de contenedores

Con la llegada de la tecnología de contenedores han surgido nuevos retos. El paradigma que promueven es de un ciclo de vida corto y sin estado, situación que provoca la necesidad de gestionar arquitecturas dinámicas, las cuales cambian sin previo aviso y de manera rápida, creando y destruyendo contenedores. Este paradigma se contrapone al modelo estático de MVs, en el cuales se crean MVs con poca frecuencia, con un ciclo de vida largo.

Este cambio de modelo provoca que se tenga que gestionar la creación y destrucción de gran número de contenedores, tarea para la cuál las herramientas clásicas no están preparadas. Para ello aparecen nuevas herramientas y métodos,

[4] `https://help.ubuntu.com/lts/serverguide/installation.html`

las cuales cubren esta carencia. Estas herramientas gestionan el ciclo de vida de los contenedores: creación, gestión y destrucción, ofreciendo tanto requerimientos funcionales cómo no funcionales.

Dentro de los requerimientos funcionales que ofrecen, se incluye la gestión completa del ciclo de vida de los contenedores en un entorno distribuido, ofreciendo facilidades a nivel de cómputo, red y almacenamiento que permiten gestionar estos conjuntos de contenedores de una forma sencilla, aportando además una capa de gestión independiente del infraestructura subyacente[17]. Entre los no funcionales, los más interesantes son elasticidad, alta disponibilidad y transparencia.

Existen múltiples alternativas compitiendo actualmente en el mercado de la gestión de contenedores. Algunas de estas soluciones son Warden[5], Swarm[6], Marathon[7], o Kubernetes[8]. Rápidamente, Kubernetes se ha establecido cómo el estándar de *facto* en el ámbito industrial, habiéndose alzado rápidamente cómo el más usado en el mercadoSin embargo, existen varias distribuciones de Kubernetes, las cuales agrupan Kubernetes con varias herramientas, ofreciendo una experiencia más completa, preparada, y accesible. Ejemplos de estas distribuciones son Openshift[9] o Rancher[10].

Kubernetes se ofrece cómo una solución de gestión de contenedores que, además de las características ya mencionadas, se centra sobre todo en dos aspectos: resiliencia y configurabilidad[18]. El funcionamiento de Kubernetes se basa en ofrecer un conjunto de primitivas, las cuales se construyen unas sobre otras, que permiten abstraernos de la infraestructura subyacente. Algunas de las abstracciones que ofrece Kubernetes son:

- **Pod**: La abstracción base, de la que cuelgan la mayoría de las demás. Un `Pod` es la unidad básica gestionable por Kubernetes. Un `Pod` es uno o más contenedores que comparten espacio de red, almacenamiento y contexto. Todos los contenedores de un `Pod` están siempre en una misma máquina física. Se podría comparar con un host lógico, y sería el equivalente a una máquina virtual. Esta abstracción nos permite ejecutar juntos servicios muy acoplados (Por ejemplo, un contenedor con nuestro servicio web y otro con la monitorización) sin romper el patrón de mantener un proceso por contenedor. Este patrón de ejecutar varios contenedores de manera acoplada se denomina "patrón sidecar".
- **ReplicaSet**: Mantiene el número de `Pods` del tipo especificado activos en todo momento, creando o destruyendo según sea necesario.

[5] `https://github.com/cloudfoundry-attic/warden`
[6] `https://docs.docker.com/engine/swarm/`
[7] `https://mesosphere.github.io/marathon/`
[8] `https://kubernetes.io/`
[9] `https://www.openshift.com`
[10] `https://rancher.com`

- **Deployments**: Ofrece unas características similares a un `ReplicaSet`, pero ofreciendo un conjunto de primitivas que nos permiten hacer operaciones más complejas. Ejemplos de estas primitivas son capacidad para aumentar/-disminuir los `Pods`, gestión de auto-escalado, y actualizaciones declarativas. Tiene la capacidad de actualizar una aplicación en ejecución "en vuelo", es decir, sin perder tráfico de red. Soporta diferentes técnicas de actualización (canario, A/B, *rolling update*).
- **DaemonSet**: Permite ejecutar `Pod` en todos los nodos del clúster (o en aquellos que cumplan cierta condición). Es usado, por ejemplo, para gestionar los `Pods` de red que han de estar en todos los nodos, o para ejecutar agentes de monitorizado con cada nodo.
- **Jobs**: Un tipo especial de `Deployment`. A diferencia de este, un `Job` representa un trabajo en *batch*, el cuál se ejecuta sólo una vez (o un número limitado). Se utiliza, por ejemplo, para tareas de mantenimiento manuales, migraciones de bases de datos, operaciones de limpieza…

Estas abstracciones se gestionan de forma declarativa, de forma que para hacer el despliegue de una aplicación se le dan al cluster uno o más manifiestos, escritos en el lenguaje YAML, los cuales determinan el estado deseado del clúster, basándose en las abstracciones ya comentadas. En ese momento comienza lo que se denomina "bucle de reconciliación". Según este modelo, Kubernetes ejecuta varios controladores, cada uno de los cuales gestiona un recurso. Estos controladores reciben la información dada por las manifiestos que se dan a Kubernetes, y entran en un bucle de preguntar al clúster el estado actual, determinar las diferencias entre el estado actual y el estado objetivo que tienen guardado, y tomar las medidas necesarias para acercarse al estado objetivo (crear/destruir recursos, cambiar configuraciones, ejecutar operaciones). Estos controladores pueden actuar en cadena, de forma que uno de ellos solicite recursos al clúster que otro vaya a procesar[11]. Es sencillo añadir nuevos controladores que interactúen tanto con los recursos ya definidos cómo con potenciales nuevos recursos que se desee añadir [19].

## 2.3 Federaciones y métodos de gestión en Kubernetes

Una posibilidad que ofrece Kubernetes, usada durante el desarrollo de este trabajo, es la de federar un conjunto de clústers, dando la opción de gestionarlos

[11]Un ejemplo de esta cadena es la línea `Deployment`→`ReplicaSet`→`Pod`. En esta cadena, cuando se solicita al clúster la creación de un `Deployment`, el `DeploymentController` crea una solicitud de `ReplicaSet` en el clúster, la cuál es atendida por el `ReplicaSetController`, el cuál hace lo mismo con los `Pods` necesarios

de manera centralizada, incluso si, cómo es el caso de este trabajo, los clústers son heterogéneos a nivel de arquitectura o tamaño.

Este modelo no es exclusivo de Kubernetes, estando relativamente extendido cómo modelo para poner en contacto varios sistemas diferentes, posiblemente de distintos dueños, ofreciendo cada uno de dichos sistemas una interfaz común, que se abstrae de las particularidades internas del sistema particular. Este paradigma permite trabajar con un conjunto de sistemas diferentes con facilidad [20].

En el caso particular de Kubernetes, el modelo de federación se ofrece no necesariamente para la gestión de sistemas diferentes, sino para la gestión de sistemas similares, pero independientes entre sí, desde un punto centralizado. Un ejemplo de la utilidad de esta funcionalidad es la de trabajar con clústers geográficamente distribuidos, para ofrecer funcionalidades de alta disponibilidad de manera transparente y sencilla.

Otro aspecto de las federaciones es la oportunidad de compartir recursos que ofrece, permitiendo tener, por ejemplo, clústers especializados, pero accesibles a todos los elementos de la federación[12]. Estos recursos pueden también no ser recursos únicos a uno de los clústers. Por ejemplo, en el sistema propuesto, el recurso primario que ofrecen los clústers de la federación es capacidad de cómputo. Como aproximación sencilla, en el modelo propuesto, los clústers ofrecen la totalidad de su capacidad de cómputo, dado que se asume que los clústers pertenecen a una misma entidad. Una alternativa sería que los clústers tuvieran una parte de capacidad privada, que sería consumido exclusivamente de manera interna por las entidades del clúster, y otra parte pública, la cuál podría ser cedida a un *pool* común, el cuál podría ser consumido por cualquier participante de la federación.

En Kubernetes, la federación es aún una opción joven, aunque ya comienzan a verse algunas de sus posibilidades, cómo la compartición de Servicios entre clústers[13], o la posibilidad de gestionar el reparto de tareas entre los distintos clústers[14] de manera precisa (por ejemplo, para imponer el reparto de trabajos según la arquitectura de los clústers, cómo es nuestro caso).

## 2.4 *Edge Computing*

Como se ha señalado anteriormente, el *Edge Computing* se presenta cómo una alternativa al *Cloud Computing*. Mientras que el *Cloud* defiende un modelo en el cual existe un sistema remoto el cuál almacena toda la información importante y el cuál procesa todo aquello que sea necesario; en un sistema *Edge*, todos los elementos tienen cierta capacidad de procesado, de tal forma que los elementos

[12] `https://Kubernetes.io/docs/tasks/federation/federation-service-discovery`
[13] `https://Kubernetes.io/docs/tasks/federation/federation-service-discovery`
[14] `https://Kubernetes.io/docs/tasks/federation/set-up-placement-policies-federation`

locales pueden procesar la información que reciben, sin necesidad de enviarla a otro lugar. Gran cantidad de los modelos de *Edge* y *Fog* siguen considerando la existencia de un *Cloud*, pero lo ven cómo un elemento de apoyo, y no el sistema primario de procesado ni almacenaje.

Este modelo tiene algunas ventajas con respecto al *Cloud*:

- **Reducción del tiempo de respuesta**: Dado que los elementos de procesado están más cercanos al usuario, este percibe un tiempo de respuesta mejor, debido a que se ahorra la latencia de red con el *Cloud*.
- **Aumento de la seguridad**: Dado que ya no es necesaria el envío de toda la información al *Cloud*, se evitan potenciales ataques de los datos en tránsito.
- **Reducción de costes**: Aprovechando mejor las capacidades de cómputo del sistema se ahorra el coste del *Cloud*, debido a que, aunque se tiene un sistema que no escala tan rápido, dicho sistema es notablemente más barato.

Pero este modelo no esta exento de dificultades:

- **Disminución de la potencia en el *Edge***: Mientras que en el *Cloud* público se considera la potencia de computado es infinita (únicamente está limitada por el coste económico), los dispositivos en el *Edge* tienden a ser elementos de menor potencia y menor capacidad.
- **Mayor dificultad de interconexión**: Al almacenar toda la información en un mismo punto, interconectar aplicaciones en un modelo *Cloud* es una operación relativamente sencilla. Por el contrario, el modelo *Edge* define el sistema cómo heterogéneo, lo que puede dificultar dicha red conexión.
- **Conexión de elementos heterogéneos**: Por definición, el *Edge* esta formado por una malla de elementos distintos, que únicamente tienen en común la capacidad de procesar información. Esto hace que sea necesario el establecimiento de algún sistema de intercambio de información común.

### 2.4.1 Scheduling en *Edge Computing*

El *scheduling* (a veces denominado "programación" en la literatura española) consiste en, dado un conjunto de elementos de cómputo y un conjunto de tareas, con ciertas limitaciones de tiempo y/o recursos, decidir cómo repartir las tareas entre los elementos de cómputo, de forma que todas las tareas puedan completarse dentro de los parámetros de tiempo/coste dados. En un sistema *Edge*, existen varios conjuntos de elementos de cómputo (cada uno de los componentes de cada *Edge*, el *Cloud*, capacidad extra en los *Edge*...) que se organizan de manera jerárquica (siendo los elementos de esa jerarquía *Cloud*, *Edge* y elementos de un *Edge*), y nuestro sistema debe poder gestionar el movimiento de tareas entre las distintas jerarquías[21][22].

Particularizando a nuestro problema, se considera un conjunto de tareas, las cuales son las imágenes a analizar, y un conjunto de sistemas de cómputo, siendo estos el *Edge* y el *Cloud*. En nuestro caso, no existen limitaciones de tiempo explícitas o *deadlines*, si no que se intenta completar las tareas en el menor tiempo posible, evitando el consumo monetario añadido de realizar el cómputo en el *Cloud*.

En la arquitectura propuesta, existen un conjunto de *Edges* y un *Cloud*, cada uno de los cuales tiene unas características particulares de latencia, potencia y coste, entre los cuales se reparten las tareas generadas. Además, existe un *scheduling* jerárquico, con un *scheduler* raíz, consistente en la federación, la cuál tiene conectividad con un *Cloud* que presenta de forma transparente un conjunto elevado de elementos de cómputo; de la misma federación cuelga un conjunto de *Edges*, cada uno de los cuales con su conjunto interno de elementos de cómputo, los cuáles puede exponer al exterior, para ser usados por la federación, de forma que los *Edges* puedan

# Capítulo 3

# *Scheduling* dinámico en entornos de *Edge computing*

Cómo ya se ha comentado durante las secciones previas, el *Edge* presenta retos específicos a nivel de *scheduling*, entre los que podemos destacar la arquitectura en malla o la poca potencia de sus dispositivos. Este capítulo presenta las formas en las que se han analizado estos problemas, y las soluciones que se proponen, además de la línea de pensamiento que se ha seguido para desarrollar dichas soluciones.

## 3.1 Descripción del Caso de Uso

El caso de uso específico que se va a considerar es el procesado de imágenes de cámaras de vigilancia en un entorno distribuido. En esta situación, se consideran una o más cámaras de vídeo, las cuales están conectadas a una o más Raspberry Pis[1], las cuales recogen las imágenes generadas por las cámaras, realizando el procesado necesario de las mismas. En nuestro caso, se plantean dos procesados necesarios:

- Primero, se comprueba si la imagen ha cambiado de forma notable con respecto a la imagen capturada previamente. Esto sirve para discriminar la necesidad de realizar el segundo paso del procesado.
- El segundo paso, realizado únicamente en el caso de que el primer paso determine que es necesario, consiste en realizar reconocimiento facial con cualquiera de las figuras detectadas en la imagen. Este paso es notablemente más exigente en potencia computacional.

[1]Una *Single Board Computer*, conocida por ser una de las primeras accesibles al público general, muy extendida entre la comunidad *hobbyist*. Se escoge estaa plataforma por razones de Precio, tamaño, y consumo

Los recursos computacionales del sistema se dividen en dos, el *Cloud* y los *Edges*, cada uno de ellos con distintas características. Generalmente, el *Edge* tendrá poca capacidad de cómputo, tiempo de respuesta rápido y coste bajo, mientras que el *Cloud* tendrá una gran capacidad de cómputo, un tiempo de respuesta lento y coste alto. Por tanto, para cualquiera de los trabajos que requiera el sistema existen dos opciones: procesarlo en el *Edge* o enviarlo al *Cloud*. La primera opción tiene las ventajas ya comentadas de precio, consumo y latencia; mientras que la segunda tiene una capacidad de procesado mayor.

Por tanto, para cada trabajo a realizar por el sistema se ha de decidir si hacerlo en el *Edge* o llevarlo al *Cloud*. Para ello, es necesario algún sistema de *scheduling*, que decida a cual de los dos lugares enviarlo, el cual tenga en cuenta aspectos como tiempo de respuesta necesario, coste o carga actual del sistema.

La arquitectura a alto nivel del sistema se presenta en la Figura 3.1. En esta arquitectura existen un conjunto de participantes en una federación, la cuál tiene conectividad con el *Cloud*. Estos participantes tendrían la posibilidad de ofrecer una parte de su potencia de computación a la federación, de forma que esta tendría la posibilidad de distribuir tareas creadas en uno de los participantes al resto. De esta forma, se crea un *pool* de potencia de computación accesible a todos los participantes, que evita tener que mandar cosas al *Cloud* en el momento que uno de ellos tenga un pico de demanda, de forma que en caso de que un *Edge* sufra uno de dichos picos, otro *Edge* con baja ocupación puede asumir dicho trabajo extra, en vez de enviarlo al *Cloud*. En cualquier caso, las tareas que se procesan se han considerado como una caja negra de elementos que llegan para ser procesado en el *Egde*, pudiendo generalizarse la arquitectura propuesta a cualquier otro ámbito de aplicación de *Egde Computing*, como puede ser el análisis de *logs*, la gestión de catástrofes, o la gestión de *smart grid*.

## 3.2 Arquitectura del sistema

La solución propuesta esta formada de 7 elementos, que se conectan de la forma descrita en la Figura 3.2

Los componentes que conforman el sistema son:

- **scheduler-cloud**: Este componente representa al *Cloud* que aparece en el diagrama del sistema (Figura 3.1). Representa la *pool* de potencia que ofrecería una *Cloud* pública/privada; en nuestro modelo, con capacidad infinita. Internamente es un *scheduler* que simplemente simula a los *workers* mediante esperas.
- **scheduler-federation**: Es el encargado de realizar el enrutado entre *Edges* y *Cloud*, según el algoritmo de *scheduling* dado. En el algoritmo testeado, es

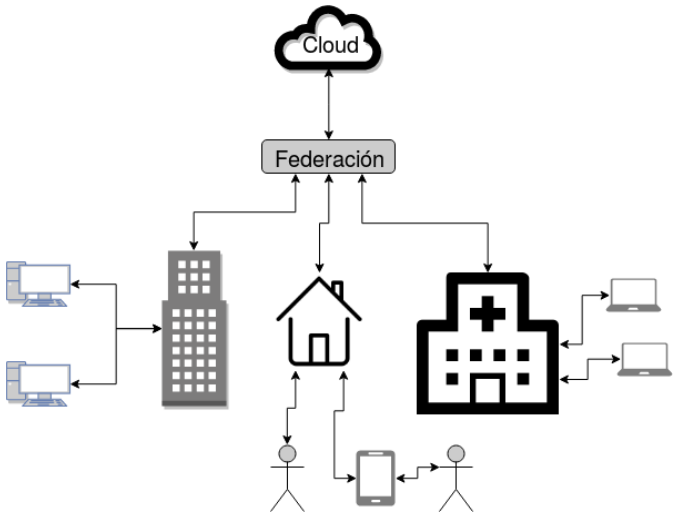



Figura 3.1: Arquitectura de alto nivel del sistema

un *Round Robin* que da prioridad a los *Edges*, con el objetivo de disminuir el coste de procesar las tareas. Internamente, esta conformado por 4 elementos:

  - ***collector***: Se encarga de recibir las tareas, realizar cualquier preprocesado necesario y de enviarlos a la cola de tareas interna.
  - ***task-queue***: Almacena las tareas a la espera de ser atendidas. En el modelo actual es una única cola tipo FIFO sin prioridades.
  - ***dispatcher***: Es el encargado de ir tomando las tareas de la cola de tareas y redirigirlas al lugar adecuado, según el algoritmo de *scheduling* escogido.
  - **monitorizado**: Es el encargado de recibir la información de carga actual de los distintos *Edges*, manteniendo el estado interno de la federación. Este estado se mantiene con un modelo de consistencia eventual.

- **task-generator**: Este componente es el elemento que introduce las tareas al sistema. Simula la generación de tareas con una distribución y frecuencia dadas y las envía al **scheduler-edge**. En los experimentos realizados, su distribución es exponencial, y su frecuencia es configurable. Genera las tareas en ráfagas, emulando la manera en la que el paso de una persona por el encuadre de una cámara generaría varias tareas, incluyendo además un *blob* de datos en la tarea, para emular de manera adecuada el comportamiento de mover las imágenes por la red. Todas las opciones son configurables, permitiendo generar un elevado rango de situaciones cambiando unas pocas opciones de configuración.
- **task-worker**: Es el encargado de realizar las tareas. Estas son un análisis de la imagen mediante la librería OpenCV, con los *bindings* de Python. Ini-

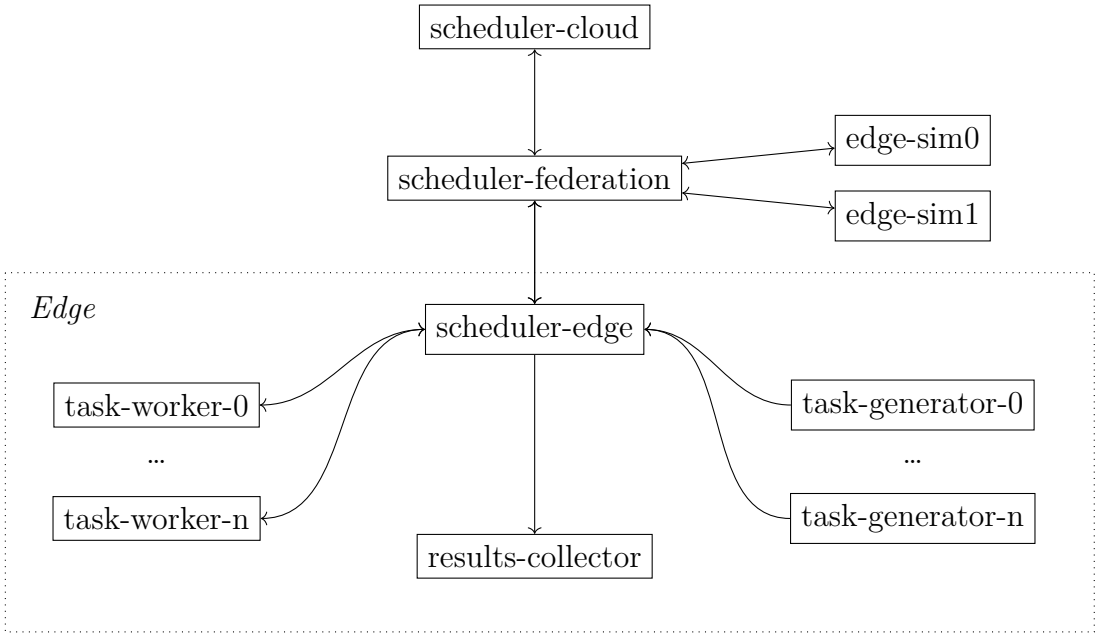



Figura 3.2: Arquitectura del modelo creado del sistema descrito.

cialmente se planteo la posibilidad de crear y destruir estos contenedores al vuelo, con cada análisis necesario. Esta opción se descarto tras los experimentos descritos en la Sección 4.2.1.

- **scheduler-edge**: Es el componente que se encarga de conectar todos los elementos en el *Edge* real. Al igual que el **scheduler-federation**, contiene los siguientes elementos:
  - ***collector***: Es el encargado de recibir las tareas, tanto internas cómo externas, preprocesarlas y enviarlas a la cola.
  - ***task-queue***: Almacena las tareas hasta que sean atendidas por el *dispatcher*. Es una única cola FIFO.
  - ***dispatcher***: Es el encargado de tomar las tareas de la cola y redirigirlas al lugar adecuado: a un *worker* interno en caso de que este libre, o a la federación en caso contrario, según el algoritmo de *scheduling* configurado. En los experimentos, el algoritmo utilizado es *Round-Robin*, dando prioridad a los *workers* internos
  - **monitorizado**: Se encarga de mantener el estado del *Edge* actualizado en la federación. Puede responder a peticiones o enviar actualizaciones periódicas. En los experimentos se usaba el segundo modo.
- **results-collector**: Es el componente que se encarga de recibir la información sobre las tareas completadas del **scheduler-edge**, presentándolas de forma

que sean analizables con comodidad.

- **edge-sim[0-1]**: Estos componentes simulan la funcionalidad de un *Edge* completo, incluyendo **task-generator**, **scheduler-edge**, **task-worker**, y **results-collector**. El objetivo de este componente es comprobar el comportamiento de el sistema en una situación en la que haya varios *Edge*, sin el consumo de recursos asociado. Simula todas las características de los *Edge*, incluyendo todos los componentes descritos, comportamiento y latencia de red, consumo de recursos...Al igual que los componentes previos, todo su comportamiento es configurable.

Todos los componentes, y especialmente aquellos formados por subcomponentes, están diseñados pensando en la escalabilidad. Por ejemplo, el *scheduler-federation* y el *scheduler-edge* utilizan una arquitectura interna generalizada, de forma que sería sencillo (Aunque requeriría usar cambios) separar los componentes en nodos independientes y replicarlos, de forma que podríamos tener varios elementos *collector*, los cuáles alimentarían todos a un sistema de colas (Cómo RabbitMQ o Kafka), de donde recogerían las tareas varios *dispatcher*, que los enviarían al lugar adecuado.

## 3.3 Implementación

Para la implementación se ha escogido el lenguaje Golang. Esto responde a su profunda relación con las tecnologías implicadas en el trabajo (Docker y Kubernetes), permitiendo tener acceso directo a gran cantidad de funcionalidades sin necesidad de librerías externas. Además, el tipo de sistema a realizar (sistema distribuido en red) es uno de los puntos fuertes del lenguaje, ofreciendo directamente a nivel de lenguaje facilidades para la gestión de hilos (mediante la primitiva `go`), la comunicación y paso de mensajes por red (mediante `gRPC`), y características de tolerancia a fallos de serie.

Dado que el trabajo se ha desarrollado bajo el sistema Kubernetes, se ha intentado en todo momento hacer el máximo uso de las abstracciones y funcionalidades que ofrecía el mismo. Modelos cómo los `Deployments`, los `Daemonsets`, o los `Jobs` permiten simplificar el despliegue de aplicaciones notablemente. Cómo ejemplo de ello, una posibilidad del sistema es el escalado de *workers* de manera sencilla, mediante una combinación de `Daemonsets` y `Labels`, la cuál permite marcar nodos como *workers* con un único comando, haciendo Kubernetes que todo se ejecutara de manera adecuada.

# Capítulo 4

# Evaluación experimental

Con el objetivo de valorar el rendimiento del sistema diseñado e implementado en el capítulo anterior, se ha elaborado una metodología experimental, consistente en el despliegue de una serie de experimentos en un entorno real. El objetivo de este capitulo es presentar de manera precisa dicha metodología. Para ello, se parte de una descripción del entorno experimental, de la propia metodología y por último, de los experimentos realizados. Adicionalmente, como elemento previo a la validación experimental, se han realizado un conjunto de experimentos preliminares que han sido utilizados para tomar determinadas decisiones a nivel de diseño.

## 4.1 Entorno experimental

El entorno de evaluación consiste en 2 máquinas de arquitectura (*x86_64*) y un clúster Kubernetes de 8 Raspberry Pis. El reparto de componentes dentro de este entorno es:

- Máquina 1: Clúster Kubernetes de una máquina, la cual simula el *Cloud*, por lo que tiene la latencia aumentada tal y cómo se describe en la Sección 4.2.3, e incluye los siguientes componentes:
  - **scheduler-cloud**

  Características:
  - **CPU**: 1xAMD Opteron(tm) Processor 146 (2GHz)
  - **RAM**: 2 GB
- Máquina 2: Clúster Kubernetes de una máquina, la cuál ejecuta los siguientes componentes:
  - **scheduler-federation**

  Características:

  - **CPU**: 2xIntel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz
  - **RAM**: 8 GB
- Máquinas 3–10 Clúster Kubernetes de 8 máquinas (1 maestro y 7 nodos), la cual simula el los *Edges*, y ejecuta los siguientes componentes:
  - **scheduler-edge**:
  - **task-worker**: 20 réplicas de este componente, conectados con el **scheduler-edge**
  - **task-generator**: 2 réplicas de este componente, conectados con el **scheduler-edge**
  - **results-collector**: 1 réplica de este componente, conectado con el **scheduler-edge**
  - **edge-sim**: Dos réplicas de este componente, denominadas *edge-sim0* y *edge-sim1*.

  Características:
  - **CPU**: 4xARMv7 Processor rev 4 (v7l) (1.2GHz)
  - **RAM**: 1 GB

Los componentes descritos se conectan físicamente de la forma cómo se describe en la Figura 4.1.

## 4.2 Experimentos preliminares

En varios momentos durante el desarrollo de este trabajo se han presentado decisiones de diseño a realizar en las cuales no era obvio la opción más adecuada. En estos puntos se ha intentado utilizar el método científico, guiando las decisiones en la medida de lo posible por experimentos realizados. En esta sección se exponen aquellos relacionados con el modelo de implementación en Kubernetes, la cantidad de tareas que procesa cada Raspberry Pi y la latencia incurrida al comunicar con el *cloud*.

### 4.2.1 Modelo basado en workers o en Jobs

Inicialmente, se planteo la posibilidad de utilizar como método de procesado de tareas el sistema de *Jobs* de Kubernetes. Este sistema permite crear tareas a realizar en el clúster, con la condición de que tienen que realizarse una única vez. Esta abstracción se opone a los *Deployments* típicos, usados más en servicios de vida larga. Algunos ejemplos de *Jobs* serían copias de seguridad, análisis de datos, envíos de correos…

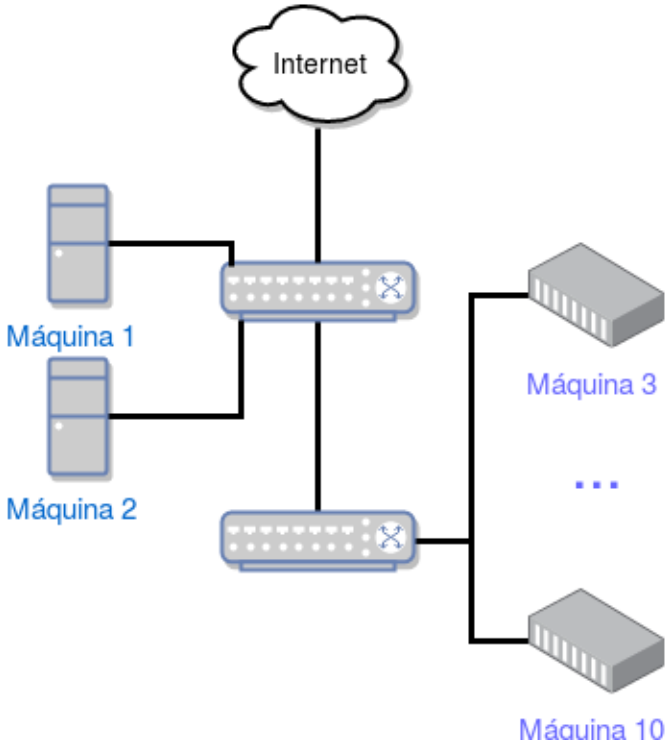



Figura 4.1: Arquitectura física del entorno experimental

Una ventaja de este modelo es el hecho de poder utilizar el scheduler de Kubernetes, en vez de implementar el nuestro propio sobre la plataforma. Otra de las ventajas es el hecho de que esa posibilidad se considera más idiomática y alineada con el modelo ofrecido por Kubernetes.

Sin embargo, al realizar la implementación inicial de dicho modelo se descubrió que, en el tipo de cluster usado (Máquinas ARM de bajo coste y potencia), el tiempo de creación, scheduling y destrucción de una tarea es muy elevado comparado con el tiempo de procesado de la tarea en sí (Los tiempos involucrados se presentan en la Figura 4.2)

Dados los resultados de este experimento, en el que se ve cómo más del 50% del tiempo de vida de la tarea se pierde en la administración del *Job*, se ha tomado la decisión de pasar a un modelo basado en *workers*, en el cuál existen un conjunto fijo de procesos ya existentes, conocidos por el *scheduler*, a los cuales se envían las tareas para su realizado.

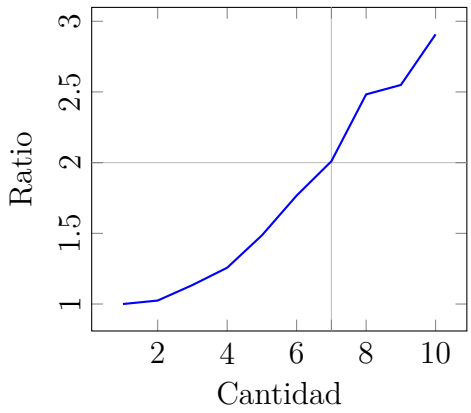
| Fase | Tiempo ($s$) |
|---|---|
| Creación del *Job* | 3.2 |
| Scheduling del *Job* | 0.7 |
| Creación del contenedor (Asumiendo precarga de imagen) | 0.7 |
| Procesado de la tarea y envío de resultados | 7.2 |
| Destrucción del *Job* | 4.7 |

Figura 4.2: Tiempos de las distintas fases del ciclo de vida de un *Job* que procesa una tarea.

### 4.2.2 Numero de workers por dispositivo

Una vez se realiza el experimento comentado en la sección 4.2.1 y se toma la decisión de continuar el trabajo con una modelo basado en *workers*, surge la pregunta de cuántos *workers* colocar en cada nodo de procesamiento disponible. Al igual que en el caso previo, se decide tomar una aproximación experimental, para comprobar como la coexistencia de varios contenedores en un mismo nodo afecta a su rendimiento.

Para ello, se determina realizar un experimento en el cuál se realicen varios procesados de una misma tarea, con distintas cantidades de tareas concurrentes en un nodo, comprobando así cómo afectan al tiempo de realización. Los resultados de este experimento se presentan en la Figura 4.3




| N° de contenedores | Tiempo de tarea ($s$) |
|---|---|
| 1 | 6.07 |
| 2 | 6.22 |
| 3 | 6.88 |
| 4 | 7.63 |
| 5 | 9.02 |
| 6 | 10.73 |
| 7 | 12.20 |
| 8 | 15.07 |
| 9 | 15.48 |
| 10 | 17.65 |

Figura 4.3: Efecto de tener múltiples tareas concurrentes procesandose en un mismo nodo. Ratios con respecto al tiempo base y valores absolutos

Es importante entender esta figura sabiendo que la Rapsberry Pi usada durante el experimentos es un sistema de cuatro núcleos y la aplicación utiliza como máximo

uno. De esa información se puede ver cómo la degradación es pequeña en el rango $[1, 4]$, causada por cambios de contexto y contención de caché (idealmente, el ratio de degradación en este rango debería ser 1), y comienza a crecer al llegar a cinco contenedores/nodo. En la gráfica se marca el valor a partir del cuál el ratio de degradación es superior a 2, siendo este 7, lo que significa que en ese punto es mejor ejecutar los contenedores de manera secuencial, en vez de concurrentemente.

Tras este experimento, se decide optar por 5 contenedores/nodo. Aunque este valor se sitúa fuera del rango ideal, permite aprovechar más el número de nodos disponibles, perdiendo únicamente un 48% del rendimiento. El tiempo medio de tarea en esta situación es $9.0217s$

### 4.2.3 RTTs con *Clouds* conocidos

Un aspecto importante que distingue al modelo basado en *Edge* del modelo *Cloud* es la latencia de red involucrada. Mientras que el uso de un *Cloud* público (Como AWS, GCP, Digital Ocean…) implica una cierta latencia de red, el uso de un *Edge* cercano, no solo lógicamente sino también físicamente hace que este aspecto desaparezca, dado que para comunicarse con el *Edge* únicamente se mueve la información a nivel de LAN.

Para hacer la simulación más realista, se ha añadido latencia extra al utilizar el *Cloud*. Para ello, se hace uso de la herramienta *tc*[1], la cuál permite modificar flujos de paquetes dentro de una máquina, permitiendo añadir latencias fijas o variables, perdidas de paquetes o implementar calidad de servicio…

Se usó cómo ejemplo la latencia de un *Cloud* público, para poder tener una base significativa. Se escoge **AWS**[2], por ser el *Cloud* público más usada en la actualidad. Habiendo escogido AWS, se realizo en una batería de pruebas consistente en la realización de *pings* a máquinas situadas en las regiones más cercanas, comprobando la latencia de las mismas.

En estas pruebas se observó que el RTT de AWS era fijo, de 108ms, por lo que, para simular correctamente dicho comportamiento, se añadió la misma latencia a la máquina en la que se ejecutaba el elemento **scheduler-cloud**, con el comando "`sudo tc qdisc add dev enp0s10 root netem delay 54ms`".

## 4.3 Metodología experimental

Para la realización de los experimentos se han diseñado *scripts* que permiten su ejecución y la recogida automática de resultados. Los experimentos se realizaron en series de 550, de los cuales se descartan para el análisis los 50 primeros. Esto se

[1] `https://wiki.debian.org/TrafficControl`
[2] `https://aws.amazon.com`

hace para permitir que el sistema entre en estado estacionario. Se realizan, a menos que se especifique lo contrario, tres series para cada experimento. Inicialmente se planteó hacer series más largas, pero tras hacer algunas pruebas se comprobó que no tenían diferencia estadística significativa.

Para los experimentos, se van a considerar 3 escenarios principales:

- **Escenario 1**: Sólo *Edge* activo. Este escenario supone el uso únicamente de la capacidad computacional disponible en el *Edge*.
- **Escenario 2**: *Edge* y *Cloud* activos. Escenario mixto que permite que se aproveche la capacidad de computación del *Edge* y se utilice el *Cloud* como soporte para incrementar la elasticidad.
- **Escenario 3**: Sólo *Cloud* activo. Este escenario supone que todas las tareas se procesan en el *Cloud*, lo que implica que las Raspberry Pis se comportan como colectores de datos y no se utiliza su capacidad de cómputo.

Para cada escenario, se variará la frecuencia de generación de tareas ($\lambda$), para poder visualizar cómo se comporta el sistema frente a distintas situaciones de carga. $\lambda$ es la frecuencia de una distribución de probabilidad exponencial, por lo que una $\lambda$=0.1 significa una generación de una ráfaga de tareas cada $\frac{1}{0.1} = 10s$ de media.

Las métricas que se van a considerar son:

- **Throughput**: Se medirá en una media tareas por segundo. No se tendrán en cuenta el tiempo entre tareas. Esto se hace para evitar que la diferencia en velocidad de generación de tareas afecte a nuestras métricas. Se medirá, a menos que se especifique lo contrario, cómo una media móvil, para evitar que variaciones amplias locales afecten a nuestras métricas. Aunque se detallará más adelante es interesante destacar que en varios de los experimentos la limitación de *throughput* ha estado en la frecuencia de generación, y no en el sistema[3].
- **Porcentaje de tareas en cada extremo**: Para cada experimento, se contabilizará que porcentaje de tareas se ha ejecutado en cada uno de los extremos posibles, esto son: *Cloud*, *Edge* propio y *Edge* externo.
- **Saltos realizados**: Métrica que indica los viajes por la red que ha realizado la tarea. Esto se medirá desde el momento en el que la tarea sale de su **task-generator** hasta que termina en su **results-collector** propio.

[3]Por ejemplo, en nuestro modelo de nube de capacidad ilimitada, el *throughput* puede crecer de manera arbitraria, dado que no esta limitado por la capacidad del sistema

## 4.4 Resultados

Los valores de $\lambda$ variaran según el escenario particular. Los valores de $\lambda$ que se tomarán para cada escenario son:

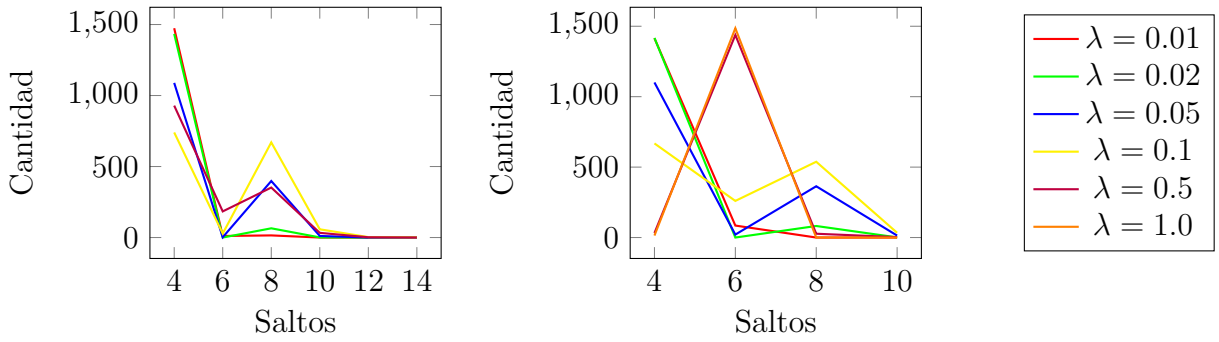
- **Escenario 1**: [0.01, 0.02, 0.05, 0.1, 0.5].
- **Escenario 2**: [0.01, 0.02, 0.05, 0.1, 0.5, 1.0].
- **Escenario 3**: [0.02, 0.5, 1.0].

Estos valores se han escogido para mostrar el rango de funcionamiento de cada escenario. En el caso del Escenario 1, no se ha incluido el valor $\lambda$=1.0 porque el *Edge* por si mismo no soporta esa carga. En el Escenario 2, se han incluido todos los $\lambda$, dado que es el más representativo. En el caso del Escenario 3 se han escogido menos valores más espaciados dado que los tiempos del *Cloud* no están afectados por $\lambda$, y por tanto no era necesario realizar la misma cantidad de experimentos.

### 4.4.1 Análisis del número de saltos

En la figura Figura 4.4 se puede ver el número de saltos que ha de realizar una tarea antes de ser completada (Consideramos un salto un viaje de red a otro de los elementos de la arquitectura, y consideramos una tarea cómo completada desde el momento que es generada hasta el momento que llega a su **results-collector** asociado). No se representa el Escenario 3 dado que en ese escenario las tareas siempre dan 6 saltos. Estas gráficas pueden entenderse mejor con la ayuda de la figura 3.2. El significado de cada valor de saltos es:

- **4**: La tarea se ha ejecutado en el *Edge* local. El camino que ha seguido es: **task-generator** →**scheduler-edge** →**task-worker** →**scheduler-edge** →**results-collector**.
- **6**: La tarea se ha ejecutado en el *Cloud* o en el *Edge* local (Tras pasar por la federación). El camino que seguiría en el primer caso es: **task-generator** →**scheduler-edge** →**scheduler-federation** →**scheduler-cloud** →**scheduler-federation** →**scheduler-edge** →**results-collector**, mientras que en el segundo este sería: **task-generator** →**scheduler-edge** →**scheduler-federation** →**scheduler-edge** →**task-worker** →**scheduler-edge** →**results-collector**
- **8**: La tarea se ha ejecutado en un *Edge* externo. El camino que ha seguido es: **task-generator** →**scheduler-edge** →**scheduler-federation** →**edge-sim** →**task-worker** →**edge-sim** →**scheduler-federation** →**scheduler-edge** →**results-collector**.
- **10 o más**: La tarea ha seguido alguno de los caminos previos, pero por errores del *scheduler* (Enviar una tarea a un *Edge* que quede ocupado mientras esta esta en su cola) ha tenido que repetir alguno de los saltos.




(1) Escenario 1: Sólo *Edge* (2) Escenario 2: *Edge* y *Cloud*

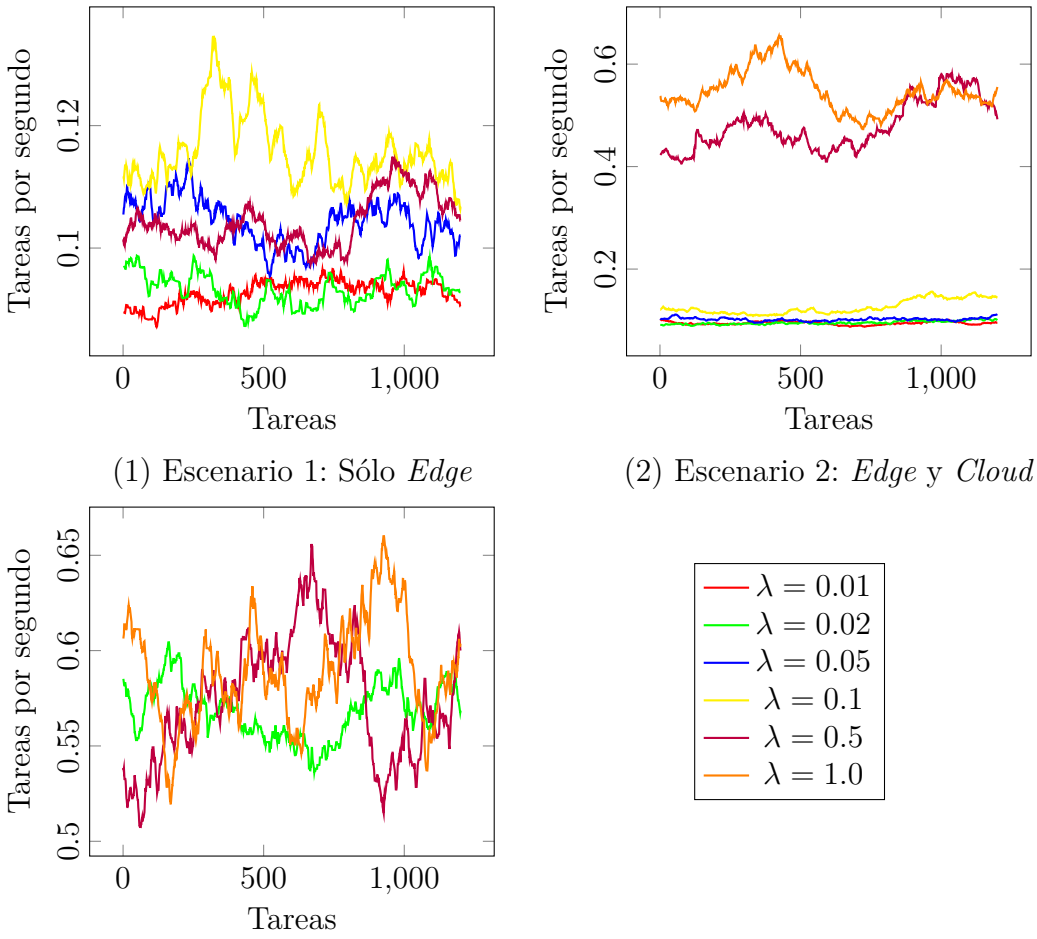
Figura 4.4: Número de saltos de cada tarea.

Al observar el Escenario 2, $\lambda$=0.1, se puede ver que el porcentaje mayor de tareas ha quedado en el *Edge* local (4 saltos), una cantidad algo menor a ha ido a un *Edge* externo, y se puede observar cómo las menos se han ejecutado en la nube. También se puede ver cómo unas pocas han dado más saltos de los necesarios, requiriendo en los experimentos como máximo 10 para llegar a completarse. Estos saltos extra son más notables en el Escenario 1, $\lambda$=0.1, donde se puede observar cómo algunas tareas han llegado a necesitar hasta 14 saltos para completarse. También se puede ver cómo algunas tareas han necesitado 6 saltos, pese a no haber *Cloud*. En este caso, las tareas han seguido el segundo caso mencionado en los 6 saltos. En la gráfica del Escenario 2 se puede ver cómo conforme aumentamos la $\lambda$, más y más tareas se desplazan al *Cloud*. El objetivo de esta gráfica es ver que la cantidad de saltos que da una tarea converge, es decir que no sufrimos inanición. En cualquier caso, tenemos un límite de 20 saltos, de forma que preveamos la inanición en casos extremos.

### 4.4.2 Análisis del throughput

En la Figura 4.5 se puede ver el *throughput* del sistema, expresado cómo una media móvil de tareas por segundo a lo largo del desarrollo del experimento. En el Escenario 3 se puede observar cómo en la nube no se existen diferencias altas de *throughput* entre las distintas $\lambda$. Esto se debe a que el *Cloud* no cambia de comportamiento dependiendo de la cantidad de tareas que se le envíe, dado que no sufre contingencia de *workers*. En el Escenario 1 se puede observar el comportamiento de *Edge* por sí mismo, viéndose que el *throughput* tiene algo de variabilidad dado el $\lambda$, pero dentro del rango [0.1, 0.12], lo cuál es una variabilidad prácticamente despreciable. El Escenario 2 es el que muestra el funcionamiento del sistema

completo, siendo soportado por *Edge* y *Cloud*. En este caso, se puede ver que el *throughput* sí depende de $\lambda$. En este caso, esta dependencia aparece por la sobrecarga del sistema. Al pasar el límite a partir del cuál el *Edge* por sí sólo no soporta la carga (Alrededor de $\lambda = 0.5$), gran cantidad de la carga se desplaza al *Cloud* (Esto se detalla mejor en la Figura 4.6), lo que provoca una mejora del *throughput* notable. Esto se puede explicar por la mayor potencia del *Cloud*, que provoca que complete las tareas a mayor velocidad, y de un *throughput* mucho mayor.




(1) Escenario 1: Sólo *Edge*

(2) Escenario 2: *Edge* y *Cloud*

(3) Escenario 3: Sólo *Cloud*

Figura 4.5: Tareas por segundo completadas en cada escenario. Las gráficas no están a la misma escala y no son comparables, para permitir tener la mayor precisión en cada una de ellas.

### 4.4.3 Análisis del uso del *Cloud*

En el caso de la Figura 4.6, se puede ver el reparto de tareas entre el $Edge_{local}$, el $Edge_{externo}$, y el *Cloud*. Se puede ver que en los casos de $\lambda$ más baja el *Edge* puede soportar la carga sin problemas siendo solo necesario el uso del *Cloud* en los de $\lambda$ mayor. En esta figura se puede ver el ahorro económico de combinar ambas soluciones de forma más acusada. Por ejemplo, tomando cómo caso el Escenario 2, $\lambda$=0.1, lo cuál representa una situación media (1 persona cada 10 segundos).

En esta situación, se puede ver cómo el reparto de tareas es relativamente equilibrado. Se puede calcular el coste aproximado de ejecutar esta operación en un sistema cómo AWS Lambda[4]. Un servicio de este estilo cobra por tiempo de tareas, por lo que, para calcular el coste completo de una ejecución usaríamos la función:

$$C_t = N_t \cdot R_c \cdot C_u$$

Siendo las variables:

- $C_t$: Coste total de las tareas
- $N_t$: Número de tareas totales
- $R_C$: Ratio de tareas enviado al *Cloud*.
- $C_u$: Coste de procesar una tarea. Vamos a considerar que el coste será 1 unidad monetaria por segundo de ejecución, por lo que si la máquina es similar a la que actúa cómo **scheduler-cloud** en nuestro entorno de pruebas, el coste de procesar una tarea será 2.5 unidades monetarias.

Por tanto, sustituyendo para nuestro caso particular:

$$C_t = N_t \cdot R_c \cdot C_u = 1500 \cdot 0.1727 \cdot 2.5 = 647.625\ u_{monetarias}$$

Si hubiéramos ejecutado todo en el *Cloud*, el coste sería:

$$C_t = N_t \cdot R_c \cdot C_u = 1500 \cdot 1.0 \cdot 2.5 = 3750.0\ u_{monetarias}$$

Se puede ver que hemos ahorrado 5.8 veces el coste por haber ejecutado gran parte de las tareas en local. Si consideramos además que un 40% de las tareas que no se han ejecutado en la nube no se han ejecutado de manera local, sino por otro *Edge* de la federación, se puede ver como el ahorro por compartir recursos es notable [5]. En definitiva, se ha reducido el coste económico en un factor de casi seis a costa de un empeoramiento del *throughput* de un factor de 4.2.

[4] `https://aws.amazon.com/lambda`

[5] Este cálculo no toma en cuenta el coste de enviar las imágenes a la nube. Considerando que muchas de las nubes públicas cobran por tráfico de red, el coste de la nube sería aún mayor.

| Escenario | $\lambda$ | $Edge_{local}$ (%) | $Edge_{externo}$ (%) | *Cloud* (%) | *Throughput* |
|---|---|---|---|---|---|
| (1) Sólo *Edge* | 0.01 | 99.00 | 1.00 | 0.00 | 0.092 |
| | 0.02 | 95.60 | 4.40 | 0.00 | 0.093 |
| | 0.05 | 72.73 | 27.27 | 0.00 | 0.104 |
| | 0.1 | 51.20 | 48.80 | 0.00 | 0.115 |
| | 0.5 | 74.20 | 25.80 | 0.00 | 0.104 |
| (2) *Edge* y *Cloud* | 0.01 | 94.33 | 0.00 | 5.67 | 0.094 |
| | 0.02 | 94.47 | 5.53 | 0.00 | 0.095 |
| | 0.05 | 73.47 | 25.20 | 1.33 | 0.101 |
| | 0.1 | 44.60 | 38.13 | 17.27 | 0.125 |
| | 0.5 | 2.20 | 2.00 | 95.80 | 0.479 |
| | 1.0 | 0.93 | 0.00 | 99.07 | 0.547 |
| (3) Sólo *Cloud* | 0.02 | 0.00 | 0.00 | 100.00 | 0.568 |
| | 0.5 | 0.00 | 0.00 | 100.00 | 0.574 |
| | 1.0 | 0.00 | 0.00 | 100.00 | 0.589 |

Figura 4.6: Porcentaje de tareas en cada extremo en cada escenario

# Capítulo 5

# Conclusión y trabajo futuro

A lo largo del trabajo se ha analizado la ventaja del uso del paradigma *Edge* en determinados contextos aplicativos, frente al uso del *Cloud*. Para aprovechar la flexibilidad de dicho paradigma, así como para aumentar la elasticidad, se ha diseñado una arquitectura de procesado basada en virtualización a través de contenedores. Además, se han mostrado empíricamente dichas ventajas en diferentes niveles, en especial el aspecto económico. Cómo se ha visto en el capítulo experimental, utilizar la computación adicional del *Edge* pueden aportar un ahorro económico importante (que puede ser incluso cinco veces menor), con un empeoramiento del rendimiento asumible en la mayoría de casos. Otra ventaja importante que aportan estos sistemas es la mejora en el aprovechamiento de los recursos del *Edge*, implicando esto mejoras económicas, ecológicas y de gestión. Tampoco se han visto gastos extra por el coste de *scheduling* ni red, aunque esto puede variar si crece mucho el sistema.

Por otra parte, la gestión de un entorno *Edge* ha supuesto trabajar en un sistema como dispositivos e infraestructura de diferentes arquitecturas. Como método de gestión y de compartición de recursos computacionales, se ha trabajado con la posibilidad de federar distintas asociaciones. Este planteamiento es extensible a entidades con mayor nivel de heterogeneidad, incluso en su origen y en su objetivo, para competir con algunos de los gigantes establecidos, como AWS y GKE, y no depender tanto de sus servicios. Estos planteamientos enlazan con los modelos de "Economía Colaborativa" que cada vez más se extienden entre la población, y poco a poco llegan a la empresa. También se proponen, cómo en el caso que se ha tratado, cómo una forma transparente de administrar sistemas heterogéneos, de forma que se disminuya notablemente los costes administrativos.

Los sistemas de *scheduling* en *Edge* abren gran cantidad de potenciales vías de investigación a partir del presente trabajo. La más evidente sería la investigación de diferentes algoritmos de *scheduling* y su evaluación. Mejoras en esta línea supondría incluir planteamientos de calidad de servicio como *deadlines*, algorit-

mos de prioridad, o algoritmos que consideren otros aspectos de las tareas. Otra posibilidad sería la generalización aún mayor del sistema; por ejemplo, aumentando la cantidad de niveles de *scheduling*, ya sea añadiendo una super-federación o añadiendo elementos por debajo del *Edge*. En línea con la posibilidad de federar empresas con interés económico, sería interesante la posibilidad de permitir a un *Edge* reservar una parte de su capacidad para uso exclusivamente privado (por ejemplo, un n% de sus workers), de forma que no pueda ocurrir el problema que un *Edge* cree tareas y todos sus *workers* propios estén ocupados con tareas ajenas. Otro aspecto relacionado sería la investigación de modelos para fomentar la participación de empresas de interés económico en este tipo de proyectos

Por último, resulta interesante señalar que, por simplicidad, no se han analizado las implicaciones a nivel de seguridad en la arquitectura propuesta. Sin embargo, si se trabaja con tareas que incluyan información potencialmente privada (Información médica o económica), sería interesante un *scheduler* que fuera consciente de este aspecto, y no enviara a terceros dichas tareas (o eliminara la información sensible previo envío)

# Bibliografía


[1] C. Weng, J. Zhan e Y. Luo. "TSAC: Enforcing Isolation of Virtual Machines in Clouds". En: *IEEE Transactions on Computers* 64.5 (mayo de 2015), págs. 1470-1482. ISSN: 0018-9340. DOI: 10.1109/TC.2014.2322608.

[2] M. Satyanarayanan. "Edge Computing". En: *Computer* 50.10 (oct. de 2017), págs. 36-38. ISSN: 0018-9162. DOI: 10.1109/MC.2017.3641639.

[3] Mohamad Nasser. *Four advantages of edge computing*. https://business.sprint.com/blog/four-advantages-edge-computing/. Accesed: 2018/06/26. Nov. de 2016.

[4] A. Yousefpour, G. Ishigaki y J. P. Jue. "Fog Computing: Towards Minimizing Delay in the Internet of Things". En: *2017 IEEE International Conference on Edge Computing (EDGE)*. Jun. de 2017. DOI: 10.1109/IEEE.EDGE.2017.12.

[5] S. Rizvi y col. "A centralized trust model approach for cloud computing". En: *2014 23rd Wireless and Optical Communication Conference (WOCC)*. Mayo de 2014. DOI: 10.1109/WOCC.2014.6839923.

[6] W. Shi y col. "Edge Computing: Vision and Challenges". En: *IEEE Internet of Things Journal* 3.5 (oct. de 2016), págs. 637-646. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2579198.

[7] K. Dolui y S. K. Datta. "Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing". En: *2017 Global Internet of Things Summit (GIoTS)*. Jun. de 2017. DOI: 10.1109/GIOTS.2017.8016213.

[8] Steven Carlini. "The Drivers and Benefits of Edge Computing". En: ed. por Schneider Electric. Accessed: 2018/06/26. Schneider Electric. 2016, págs. 1-8.

[9] B. Parák y col. "Public-Private Cloud Federation Challenges". En: *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. Dic. de 2015, págs. 512-515. DOI: 10.1109/UCC.2015.91.

[10] C. Pahl y col. "Cloud Container Technologies: a State-of-the-Art Review". En: *IEEE Transactions on Cloud Computing* (mayo de 2017). ISSN: 2168-7161. DOI: 10.1109/TCC.2017.2702586.

[11] Abhishek Verma y col. “Large-scale cluster management at Google with Borg”. En: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[12] David K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015, All. URL: http://www.oreilly.com/webops-perf/free/kubernetes.csp.

[13] Shannon Meler. *IBM Systems Virtualization: Servers, Storage, and Software*. http://www.redbooks.ibm.com/redpapers/pdfs/redp4396.pdf. Accesed: 2018/06/25. Abr. de 2008.

[14] Poul-Henning Kamp y Robert N. M. Watson. *Jails: Confining the omnipotent root*. https://docs.freebsd.org/44doc/papers/jail/jail.html. Accesed: 2018/01/10. 2000.

[15] Daniel Price y Andrew Tucker. *Solaris Zones: Operating System Support for Consolidating Commercial Workloads*. https://www.usenix.net/legacy/events/lisa04/tech/full_papers/price/price.pdf. Accesed: 2018/01/10. 2004.

[16] W. Felter y col. “An Updated Performance Comparison of Virtual Machines and Linux Containers”. En: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On* (2015). Accesed: 2018/01/10, págs. 171-172. URL: https://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/%5C$File/rc25482.pdf.

[17] Luiz Andre Barroso y Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 1st. Morgan y Claypool Publishers, 2009. ISBN: 159829556X, 9781598295566.

[18] The Kubernetes Authors. *Kubernetes Documentation*. https://kubernetes.io/docs/home/. Accesed: From 2017/11/03 to. 2017.

[19] Ian Lewis y David Oppenheimer. *Advanced Scheduling in Kubernetes*. http://blog.kubernetes.io/2017/03/advanced-scheduling-in-kubernetes.html. Accesed: 2017/11/29. Mar. de 2017.

[20] Theodore Zahariadis y col. *Towards a Future Internet Architecture*. Springer, 2011.

[21] Rajrup Ghosh y Yogesh Simmhan. “Distributed Scheduling of Event Analytics across Edge and Cloud”. En: volume (dic. de 2017).

[22] D. Hoang y T. D. Dang. “FBRC: Optimization of task Scheduling in Fog-Based Region and Cloud”. En: *2017 IEEE Trustcom/BigDataSE/ICESS*. Ago. de 2017, págs. 1109-1114. DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.360.