Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Despliegue y monitorización de un clúster Mesos

Trabajo Fin de Máster

Máster Universitario en Ingeniería Informática

Autor: Sergio López Huguet

Tutores: Ignacio Blanquer Espert

Germán Moltó Martínez

Curso: 2016 - 2017

Resumen

Cloud computing es un nuevo paradigma que tiene como objetivo ofrecer una serie de servicios en los que los usuarios solo tienen que pagar por los recursos que utilicen y durante el tiempo en que lo hagan. Para ello, los proveedores de recursos en la nube deben poder aumentar o disminuir su infraestructura virtual dinámicamente en función la carga de trabajos, lo cual se define como elasticidad horizontal.

La elasticidad horizontal es adecuada cuando el problema que se resuelve es inherentemente paralelo. Sin embargo, cuando el problema no puede beneficiarse de un aumento en el número de recursos, se debe considerar otro tipo de elasticidad. La elasticidad vertical tiene como objetivo variar dinámicamente los recursos asignados a cada servicio en función de la calidad de servicio (*QoS)*.

El presente Trabajo de Fin de Máster utiliza el sistema de gestión de recursos Apache Mesos, cuya finalidad es la ejecución de servicios mediante aplicaciones distribuidas o *frameworks* y el control de recursos tales como almacenamiento, CPU y memoria en un conjunto de nodos computacionales.

En función de las características del *framework*, es posible variar la asignación de recursos asociados a cada servicio. Es usual que estos servicios se ejecuten en contenedores Docker o en los contenedores nativos de Mesos, ya que permiten al desarrollador encapsular las dependencias de forma que sean portables y ejecutables en todo tipo de máquinas con el único requisito de tener un gestor de contenedores compatible con Docker. El desarrollo planteado en este trabajo se basará en contenedores Docker, que pueden ejecutarse a través de los *frameworks* en Mesos.

Así, el trabajo plasmado en este documento se en centra diseñar e implementar un sistema que, dada una especificación de un trabajo encapsulado en un contenedor Docker y una *QoS*, sea capaz de desplegarlo utilizando un *framework* de Mesos (los elegidos en este trabajo son Chronos o Marathon), monitorizarlo y variar los recursos asignados con el objetivo de cumplir con la calidad de servicio acordada.

Es importante destacar que todo el desarrollo se ha llevado a cabo dentro del grupo de investigación *Grid y Computación de Altas Prestaciones* (GRyCAP) de la UPV, como parte del proyecto de investigación EUBra-BigSEA.

Palabras clave: *Cloud, Mesos, Monitorización, Docker, Calidad de servicio, Elasticidad vertical.*

Abstract

Cloud computing is a new paradigm whose aim is to offer a sort of services for which the users only have to pay for the resources they use and during the time they are using them. For that, cloud resources suppliers must be able to increment or decrement their virtual infrastructures dynamically according to the workload, what is defined as horizontal elasticity.

Horizontal elasticity is appropriate when the problems that are solved are inherently parallel. However, when the problem cannot benefit from an increase of the amount of resources, another type of elasticity must be considered. Vertical elasticity consists of varying dynamically the resources assigned to each service, according to the Quality of Service (*QoS*).

This MSc. Thesis uses the resource management system Apache Mesos, whose objective is to execute services by using distributed applications or frameworks and controlling resources such as storage, CPU and memory in a collection of computational nodes.

Depending on the framework features, it is possible to vary the resources associated with each service. It is common that these services are executed in Docker containers or Mesos native containers, as they allow the developer to encapsulate the dependencies in such a way that they are portable and executable in any machine, with the only requirement of having a containers manager that is compatible with Docker. The development proposed in this work is based on Docker containers, which can be executed through Mesos frameworks.

Thereby, the work explained along this document is focused on designing and implementing a system that, given a work specification encapsulated in a Docker container and a *QoS*, is able to deploy it using a Mesos framework (Chronos or Marathon are the used ones in this work), monitor it and vary the assigned resources with the objective of achieving the agreed *QoS*.

It is important to highlight that all the development has been performed in the research group *Grid y Computación de Altas Prestaciones* (GRyCAP) of the UPV, as part of the research project EUBra-BigSEA.

Keywords: *Cloud, Mesos, monitoring, Docker, quality of service, vertical elasticity.*

Tabla de contenidos

Capítulo 1. Introducción 9
1.1 Motivación 9
1.2 Objetivos 10
1.3 Marco de trabajo 10
1.4 Metodología empleada 10
1.5 Estructura del trabajo 11
Capítulo 2. Contexto tecnológico 13
2.1 Estado del arte 13
2.2 Tecnologías empleadas 14
2.2.1. Docker 14
2.2.2. Apache Mesos 15
2.2.3. Marathon 17
2.2.4. Chronos 18
2.2.5. Monasca Openstack 19
2.2.6. CRIU 22
2.2.7. Grafana Labs 22
2.3 Entorno de desarrollo 23
2.3.1. Python 23
2.3.2. REST 23
2.3.3. JSON 24
2.3.4. GitHub 24
Capítulo 3. Desarrollo del proyecto 25
3.1. Diseño 25
3.1.1. Arquitectura del sistema 25
3.1.2. Lanzador 27
I. Chronos 27
II. Marathon 28
3.1.3. Supervisor 29
I. Chronos 30
II. Marathon 31
3.1.4. Plugin Docker para el agente de Monasca 33

3.2. Implementación 34
3.2.1. Restricciones 34
3.2.2. ¿Cómo adaptar una aplicación? 35
3.2.3. Utilización de los componentes 35
Capítulo 4. Casos de estudio 38
4.1. Aplicación seleccionada 38
4.2. Chronos 38
4.3. Marathon 40
Capítulo 5. Resultados y discusión 41
5.1. Caso I: 41
5.2. Caso II 44
5.3. Caso III 45
Capítulo 6. Conclusiones 48
6.1. Conclusiones 48
6.2. Valoración personal 49
6.3. Trabajo futuro 49
Capítulo 7. Bibliografía 50

Tabla de figuras

Figura 1: Diagrama de Gantt el proyecto....12
Figura 2: Comparación entre los contenedores y máquinas virtuales. Fuente: [1] 14
Figura 3: Arquitectura de Apache Mesos. Fuente: [3] 15
Figura 4: Ejemplo de oferta de recursos. Fuente: [3] 16
Figura 5: Ciclo de vida de una aplicación en Marathon, donde i,r y h representan, respectivamente, el número de instancias solicitadas, ejecutándose y en estado ok. Fuente: [4] 17
Figura 6: Sintaxis de una expresión sin sub-expresiones dentro. Fuente: [7] 20
Figura 7: Arquitectura de Monasca OpenStack. Fuente: [6] 20
Figura 8: Cuadro de mando creado para visualizar datos de las aplicaciones ejecutadas utilizando Marathon. 22
Figura 9: Arquitectura del sistema para aplicaciones que utilizan el framework Chronos. 26
Figura 10: Arquitectura del sistema para aplicaciones que utilizan el framework Marathon. 27
Figura 11: Ejemplo JSON recibido de una petición /initTask para una aplicación Chronos. 30
Figura 12: Ejemplo JSON recibido de una petición /updateTask para una aplicación Chronos. 30
Figura 13: Ejemplos de peticiones que realiza el programa cuando se inicia (imagen superior) y se termina (imagen inferior) la ejecución de una aplicación Marathon. 31
Figura 14: Fichero utilizado que contiene los ticks consumidos por el contenedor con identificador $CONTAINER_ID. Fuente:[17] 33
Figura 15: Función modificada del plugin Docker del agente de Monasca. Fuente: [18] 34
Figura 16: Ejemplo de especificación del nivel de calidad para una aplicación. 35
Figura 17: JSON que define las credenciales de Chronos y Marathon. 36
Figura 18: JSON que define las credenciales de KeyStone y Monasca. 36
Figura 19: Ejemplo de especificación de un trabajo Chronos utilizado en el sistema. Concretamente, esta especificación corresponde a la aplicación openmLinpack1 del caso de estudio I. 39
Figura 20: Ejemplo de especificación de trabajo Marathon utilizado en el sistema. Concretamente, esta especificación corresponde a la aplicación openmLinpack1 del caso de estudio III. 40
Figura 21: Diferencia entre la predicción calculada y el plazo que debe cumplirse en el caso I. 42
Figura 22: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack1. 42
Figura 23: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack2 43
Figura 24: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack3 43
Figura 25: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack4. 43

Figura 26: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack5..... 44
Figura 27: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack6. 44
Figura 28: Diferencia entre la predicción calculada y el plazo que debe cumplirse en el caso II. 45
Figura 29: Porcentaje de progreso en función del instante de tiempo en el caso III. Cuando el valor se encuentra en la región amarilla o roja, se produce una disminución o aumento, respectivamente, de los recursos asignados..... 46
Figura 30: Comparación entre el tiempo consumido real y deseado de CPU en el caso III. 46
Figura 31: Representa el valor de CPU asignado a la aplicación en el instante de tiempo que acaba de trascurrir para el caso III. 47

Capítulo 1. Introducción

1.1 Motivación

Cloud computing es un nuevo paradigma que tiene como objetivo ofrecer una serie de servicios en los que los usuarios solo tienen que pagar por los recursos que utilicen y durante el tiempo en que lo hagan. Los proveedores de recursos en la nube deben disponer de una infraestructura lo suficientemente grande como para poder dar servicio a todos los clientes. De esta forma se ofrece a los usuarios la sensación de que los recursos son "ilimitados". Esto permite a los usuarios aumentar o disminuir su infraestructura virtual dinámicamente en función la carga de trabajos (*elasticidad horizontal*).

La elasticidad horizontal es adecuada cuando el problema que se resuelve es inherentemente paralelo. Por otro lado, cuando el problema no puede beneficiarse de un aumento en el número de recursos, se puede considerar la *elasticidad vertical*. Este tipo de elasticidad consiste en el aumento o disminución de los recursos asignados a un determinado trabajo en función de la calidad de servicio (*Quality of Service*, *QoS*) que están ofreciendo al cliente.

Por lo tanto, resulta interesante el desarrollo de un sistema que se encargue de monitorizar y variar dinámicamente los recursos asignados a cada servicio (*elasticidad vertical)* para cumplir con una determinada calidad de servicio.

La gestión de infraestructuras requiere el lanzamiento remoto de trabajos, la monitorización de los recursos y trabajos, el equilibrado de la carga, la gestión de la conectividad, etc. Estos servicios los proporcionan gestores de recursos de diferente tipo, atendiendo a la naturaleza de los recursos compartidos y al tipo de trabajos.

El presente trabajo fin de máster utiliza el sistema de gestión de recursos Apache Mesos, cuya finalidad es la ejecución de trabajos, principalmente trabajos en contenedores y el control de recursos tales como almacenamiento, CPU y memoria en un conjunto de nodos computacionales.

La ejecución de servicios en Mesos se hace utilizando aplicaciones distribuidas o *frameworks*. Mesos asigna a cada *framework* una cantidad de recursos y, en función de sus características, es posible variar la asignación de recursos. Existen frameworks para muchos tipos de servicio: análisis de datos (por ejemplo, Spark), servicios de larga duración (por ejemplo, Marathon), Batch Scheduling (por ejemplo, Chronos), almacenamiento distribuido (por ejemplo, Cassandra) o Machine learning (por ejemplo, TFMesos).

Actualmente, la ejecución de servicios se hace en contenedores Docker o en los contenedores nativos de Mesos. Estos permiten encapsular las dependencias de forma que sean portables y ejecutables en todo tipo de máquinas con el único requisito de tener un gestor de contendores compatible con Docker. El presente trabajo se centrará en contendores Docker, que pueden ejecutarse a través de los *frameworks* en Mesos.

Gracias a que los trabajos están encapsulados dentro de contenedores Docker, que disponemos de *frameworks* en Mesos que pueden ejecutarlos y que queremos garantizar una calidad de servicio expresada en un determinado alcance del proceso o de una determinada asignación de CPU, nace la idea de desplegar, monitorizar y variar dinámicamente los recursos asociados a cada contenedor utilizando estos *frameworks*.

1.2 Objetivos

Los objetivos de este trabajo son:

- Identificar una herramienta de monitorización que pueda integrarse fácilmente en Mesos.
- Ejecutar trabajos encapsulados en contenedores Docker en Chronos y Marathon.
- Ser capaces de monitorizar los trabajos desplegados en Mesos, así como los contenedores donde se ejecutan.
- Ser capaces de variar los recursos asignados a cada contenedor Docker.
- Implementar un entorno que, dada una especificación de un trabajo encapsulado en un contenedor Docker y una QoS, sea capaz de desplegarlo, monitorizarlo y variar los recursos asignados con el objetivo de cumplir con la calidad de servicio acordada.

1.3 Marco de trabajo

El presente trabajo forma parte de un proyecto de colaboración entre Europa y Brasil: EUBra-BigSEA [1]. El objetivo de este proyecto es ofrecer una gran plataforma *cloud* integrada, elástica y dinámica y de análisis de datos con la que enfrentarse a los grandes retos de la gestión de los datos: volumen, velocidad, variedad, seguridad, privacidad o QoS.

Concretamente, este trabajo pertenece al objetivo de cumplir con la calidad de servicio en la ejecución de trabajos, enmarcado dentro de las tareas asignadas al grupo de Grid y Computación de Altas Prestaciones de la Universitat Politècnica de València, donde se ha desarrollado este proyecto.

1.4 Metodología empleada

El desarrollo de este proyecto se ha realizado con una filosofía orientada al pensamiento ágil, definiendo los objetivos que la implementación debía cumplir y completándolos en distintas etapas. Por ello, la metodología utilizada en este proyecto es el marco de trabajo SCRUM.

[1] Sitio web del proyecto EUBra-BigSEA: http://www.eubra-bigsea.eu

Según [12], en SCRUM existen varios roles principales: *Product Owner*, *Scrum Master* y el *equipo de desarrollo*. En este proyecto, los dos primeros roles han sido desempeñados por los tutores y, el rol del *equipo de desarrollo*, ha sido desempeñado por el autor de este trabajo (con la ayuda, cuando fue necesaria, de los tutores).

Product Owner es el responsable de maximizar el valor del producto y el trabajo del *equipo de desarrollo*. Además, es el único responsable de gestionar el *backlog*. Este documento incluye todas funcionalidades y características del producto que se va a implementar.

Scrum Master es el responsable de asegurarse que la metodología está entendida y se aplica. Además, ayuda al *equipo de desarrollo* eliminando los impedimentos y proporciona los recursos necesarios.

El *equipo de desarrollo* se encarga de la implementación del producto por medio de *sprints* o paquetes de trabajo. El *sprint* está formado por la planificación, las reuniones diarias al inicio del día o *daily scrums*, el trabajo de desarrollo, la revisión del *sprint* y la retrospección sobre el *sprint* para crear un plan de mejora que aplicar durante el siguiente *sprint*.

La Figura 1 representa la planificación temporal de todos los *sprints* y tareas realizados en este proyecto.

1.5 Estructura del trabajo

En el primer capítulo, se expone la motivación y marco de trabajo del proyecto, se describe la metodología utilizada y se definen los objetivos a conseguir.

En el segundo capítulo se describe el estado del arte, el entorno de desarrollo y las tecnologías empleadas.

El tercer capítulo detalla todo el desarrollo del proyecto. La primera parte corresponde con el diseño: arquitectura del sistema y descripción de los componentes desarrollados. La segunda parte está compuesta de las limitaciones del sistema, la interfaz del usuario y cómo conseguir adaptar una aplicación para que pueda aprovechar la implementación realizada.

Los capítulos cuarto y quinto corresponden con una descripción de los casos de estudio y un estudio sobre los resultados obtenidos.

El último capítulo presenta las conclusiones obtenidas, las aportaciones realizadas y el trabajo futuro.

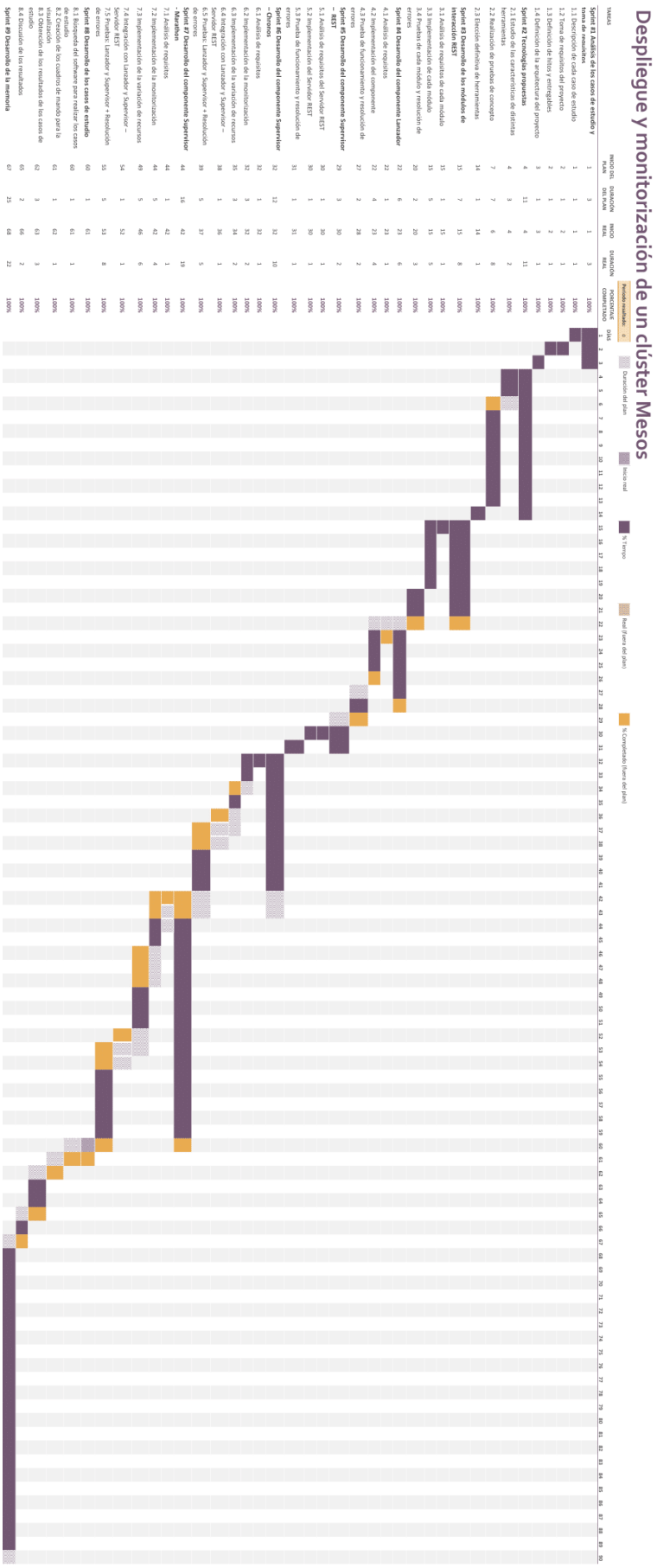

Figura 1: Diagrama de Gantt el proyecto

Capítulo 2. Contexto tecnológico

En esta sección se realizará un repaso del estado del arte y se describirán tanto las tecnologías empleadas como el entorno de trabajo donde se ha desarrollado este Trabajo Final de Máster. Además, se justificará el uso de las herramientas.

2.1 Estado del arte

La *elasticidad vertical* en *cloud computing* es un concepto ampliamente estudiado en los últimos años. Al principio del trabajo, se realizó una investigación con el objetivo de encontrar los últimos trabajos, que no hayan sido desarrollados en el proyecto EUBra-BigSEA, sobre escalabilidad vertical en *frameworks* de Mesos.

El único trabajo[2] encontrado está desarrollado por DC/OS[3] y combina contenedores Docker desplegados mediante el *framework* Marathon de Mesos con la elasticidad horizontal. El objetivo de este trabajo es realizar el aumento o la disminución de instancias en función de un objetivo específico. Consta de tres ejemplos:

- *Microscaling*: ofrece elasticidad horizontal con el objetivo de mantener el número de trabajos almacenados en una cola de *Azure Storage Queue*.
- *Requests per second* (RPS): ofrece elasticidad horizontal con el objetivo de mantener un número de solicitudes por segundo en cada servicio en función de la métrica RPS que obtiene de todas las instancias de HAProxy.
- *CPU and Memory*: aumenta el número de instancias en función del porcentaje de uso de la memoria y el tiempo de CPU consumidos por el contenedor.

Pese a que este trabajo no tiene como objetivo ofrecer una calidad de servicio mediante la escalabilidad horizontal, es interesante para este proyecto. La razón está en que, en el último ejemplo, monitoriza parámetros de los contenedores para decidir si realiza el escalado.

Dentro del proyecto EUBra-BigSEA se ha trabajo en la *escalabilidad vertical*, pero para el *framework* Spark de Mesos y sobre máquinas virtuales. Como Spark publica el porcentaje de progreso, se puede tomar fácilmente la decisión de aumentar o disminuir los recursos asociados a una máquina virtual.

[2] PoC sobre el autoescalado en Marathon: https://dcos.io/docs/1.7/usage/tutorials/autoscaling/
[3] Sitio web de DC/OS: https://dcos.io/

2.2 Tecnologías empleadas

2.2.1. Docker

Docker[4] es una plataforma de software que permite encapsular aplicaciones y sus dependencias en unidades independientes llamadas *contenedores*.

Típicamente, se definen los contenedores como *máquinas virtuales ligeras* debido a que comparten el núcleo del sistema operativo (SO) con la máquina dónde está ejecutándose, aunque realmente son grupos de procesos que se ejecutan de forma aislada y controlada sobe un sistema de ficheros independiente y virtualizado.

Gracias a esto, no es necesario virtualizar completamente el SO dando como resultado un menor consumo de recursos, mayor ligereza y un despliegue mucho más rápido.

Utilizar contenedores para el despliegue de servicios ofrece muchos beneficios ya que es posible empaquetar tanto la aplicación como todas las dependencias dentro de un mismo contenedor. De esta manera, se puede exportar el entorno de trabajo a cualquier SO y, si tiene Docker instalado, ejecutarse sin ningún problema. Por este motivo, los contenedores Docker son una excelente herramienta para la entrega de aplicaciones.

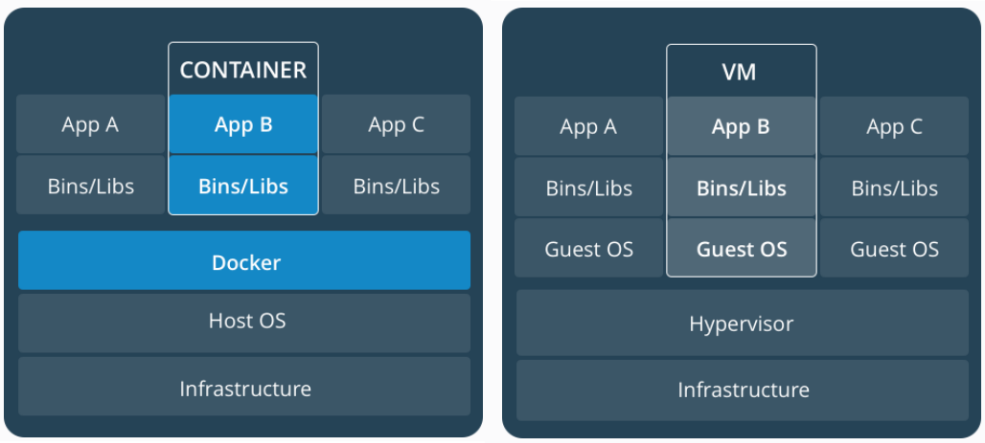

Figura 2: Comparación entre los contenedores y máquinas virtuales. Fuente: [1]

Existen varias formas para distribuir los contenedores:

- **Dockerfile**: documento en el cual se especifica la configuración del contenedor. Para poder usar la imagen resultante es necesario generarla a partir de este fichero.
- **Repositorio de imágenes**: almacenamiento donde los usuarios comparten comparte las imágenes de los contenedores para que puedan ser descargados. El repositorio utilizado en este proyecto es Docker Hub[5].

[4] Sitio web de Docker: https://www.docker.com/
[5] Sitio web Docker Hub: https://hub.docker.com/

En resumen, Docker facilita la ejecución de cualquier aplicación en un computador sin tener que instalar sus dependencias. Por esta razón, se ha decidido aprovechar esta característica para distribuir los trabajos en los nodos del clúster Mesos.

2.2.2. Apache Mesos

Apache Mesos[6] es una herramienta para la gestión de recursos de cómputo que ofrece un particionado de recursos eficiente y la compartición de recursos entre distintas aplicaciones o *frameworks* (lista completa disponible en [2]). Una infraestructura Mesos está formado por dos componentes básicos: nodos maestros (*master*) y nodos agentes (*agents*).

El nodo *master* tiene como objetivo gestionar los demonios *agents* en los nodos del clúster. Los *frameworks* ejecutan los trabajos en esos agentes.

Los agentes se registran y "ofrecen" sus recursos al nodo *master* para que este los asigne a los diferentes *frameworks* (paso 1 de la Figura 4). Esta asignación de recursos se hace dependiendo de la política de organizativa seleccionada elegida: equitativa (*fair sharing*) o con prioridad.

A su vez, los *frameworks* están formados por dos componentes: planificador (*scheduler)* y ejecutor (*executors*).

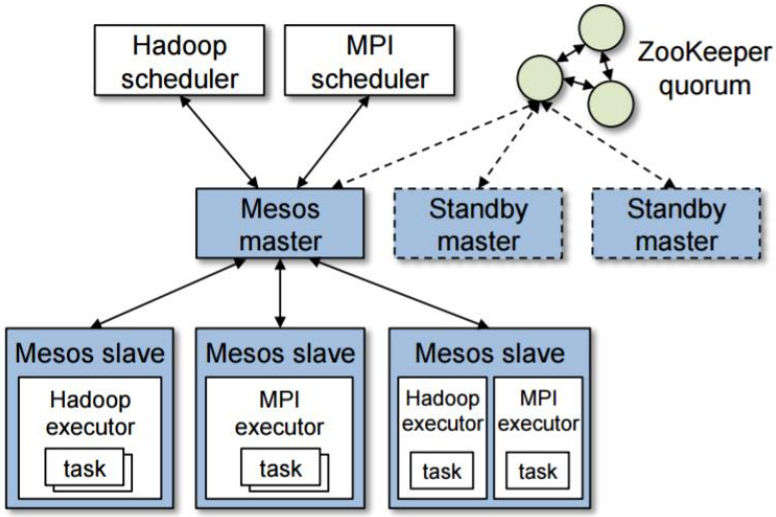

Figura 3: Arquitectura de Apache Mesos. Fuente: [3]

El *scheduler* se registra en el nodo *master* para recibir las ofertas con los recursos disponibles en los agentes (paso 2 de la Figura 4). Una vez recibe la oferta, decide cuántos de esos recursos va a utilizar y se los asigna a sus *executors*. Esta información se la envía al nodo *master* (paso 4 de la Figura 4). A continuación, el nodo *master* inicia las tareas en los agentes correspondientes (paso 4 de la Figura 4).

[6] Sitio web de Apache Mesos: http://mesos.apache.org/

Un *executor* se encarga de gestionar las tareas de los *frameworks* dentro de los nodos agentes.

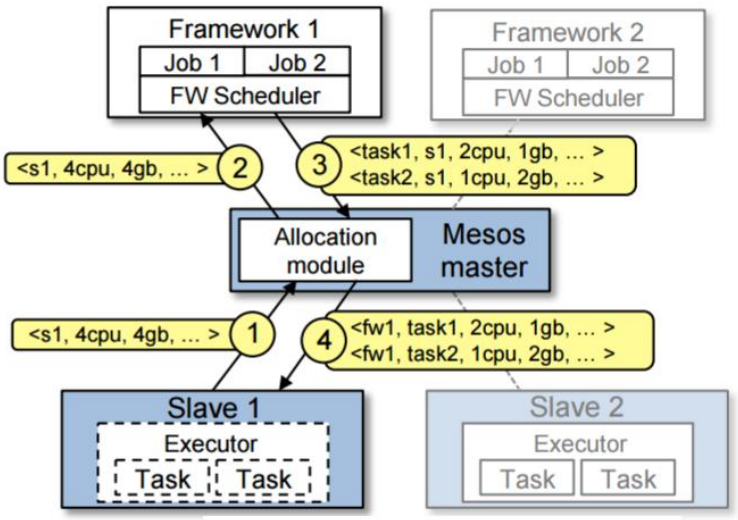

Figura 4: Ejemplo de oferta de recursos. Fuente: [3]

Si el lector quiere profundizar más en la arquitectura de Apache Mesos puede consultar la referencia [3].

Gracias a esta arquitectura, Mesos ofrece a sus usuarios:

- Escalabilidad lineal: es fácil escalar hasta 10000 nodos.
- Alta disponibilidad: Mesos es tolerante a fallos tanto en nodo *master* como en los nodos agentes utilizando Zookeeper. En el caso del nodo *master,* se utilizan deberán desplegar varios nodos. En el caso del nodo agente, si el *framework* lo permite, Mesos realiza *checkpointing* de las tareas cuando falla el nodo agente o cuando falla el *framework*.
- Contenedores: permite la ejecución de Docker e imágenes con la especificación AppC[7] de manera nativa.
- Monitorización: tanto el nodo *master* como los agentes publican una serie de estadísticas y métricas sobre los recursos utilizados y disponibles, los *frameworks* registrados, los agentes activos y el estado de las tareas.
- Aislamiento de recursos (CPUs, memoria, almacenamiento, puertos, GPUs y aislamientos personalizables) tanto entre *frameworks* como entre trabajos.
- REST APIs para el desarrollo de nuevos *frameworks* y la gestión del clúster.
- Interfaz web: permite la visualización del estado del clúster y navegar por los *sandboxes* de todos los contenedores (en ejecución y ya ejecutados).

[7] Github de App Container: https://github.com/appc/spec/

2.2.3. Marathon

Marathon[8] es un *framework* de Mesos que actúa como *Platform as a service* (PaaS). Marathon está diseñado para gestionar servicios de larga duración (que deben estar permanentemente activos) y que requieran alta disponibilidad, ya que es tolerante a fallos.

Marathon consigue la alta disponibilidad gracias que comprueba cada cierto periodo de tiempo el estado de todas las tareas. Cuando se crea una aplicación, el usuario también puede definir sus propios métodos para comprobar el estado en el apartado *healthCheck*.

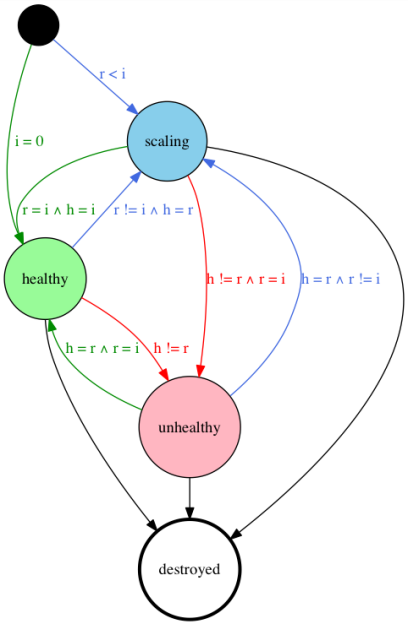

Figura 5: Ciclo de vida de una aplicación en Marathon, donde i,r y h representan, respectivamente, el número de instancias solicitadas, ejecutándose y en estado ok. Fuente: [4]

Algunas características interesantes son la posibilidad de tener varias instancias de la aplicación, la creación de volúmenes de almacenamiento persistentes, una gran API REST[9], una interfaz web muy intuitiva, la posibilidad de que las aplicaciones sean elásticas, se pueden crear *healthChecks* personalizadas para cada aplicación, permite el agrupamiento de aplicaciones, la obtención de métricas, la posibilidad suscrición a eventos o el *checkpointing* de aplicaciones cuando el planificador de Marathon o el agente de Mesos fallen.

Los trabajos pueden ser tanto contenedores Mesos (comandos *inline* o ejecución de scripts) como contenedores Docker. Se pueden definir aplicaciones de dos formas: interfaz web o API REST. La interfaz web dispone de un menú donde rellenar los campos necesarios para configurar un trabajo.

[8] Sitio web de Marathon: https://mesosphere.github.io/marathon/
[9] API REST de Marathon disponible en: https://mesosphere.github.io/marathon/docs/rest-api.html

En el caso del API REST, los trabajos se definen mediante el formato *JavaScript Object Notation* (*JSON*). Este JSON describe la especificación de hardware, el código que va a ejecutarse y más información adicional como los volúmenes, la definición de los puertos, ficheros que descargar desde enlaces, dependencias, restricciones (por ejemplo, *hostname=server1*) etc.

Debido a que Marathon permite manterner activos los trabajos sin límite temporal, la variación de los recursos asignados en las aplicaciones, la ejecución de contenedores Docker con volúmenes de almacenamiento y que dispone de una completísima API REST, se ha decidido utilizar para este trabajo.

2.2.4. Chronos

Chronos[10] es un planificador distribuido y tolerante a fallos de Mesos diseñado para la ejecución de tareas periódicas. Chronos planifica los trabajos utilizando el estándar ISO 8601 ([5]) para representar la periodicidad del trabajo. Además de trabajos que se ejecutan infinitamente, Chronos permite la ejecución de un trabajo un número finito de veces. Por ejemplo, ejecutar diez veces un *benchmark* con una separación entre ejecuciones de un minuto.

El comportamiento de este *framework* es muy parecido al del servicio *cron* de Linux, pero dispone de algunas mejoras. Por ejemplo, permite crear trabajos dependendientes de otros, es decir, que un trabajo sea ejecutado siempre y cuando se hayan ejecutado los otros trabajos anteriormente. Por ejemplo, un trabajo que realice un procesamiento de datos que necesita que los datos almacenados por otros trabajos, entonces es necesario que el procesamiento se realice después de que todos hayan guardado la información. La solución es que el trabajo que realiza el procesamiento sea dependiente de los otros, obteniendo la seguridad de que no se ejecutará hasta que todos los otros terminen.

Al igual que Marathon, Chronos puede ejecutar comandos *inline*, scripts y trabajos encapsulados en un contenedor Docker. El formato en el que se especifican los trabajos es un documento JSON. Chronos ofrece un API REST[11] y una interfaz web. Además, permite el redimensionamiento de los recursos, la definición de restricciones, la capacidad de descargar archivos al inicio de cada ejecución, etc.

Uno de los objetivos de este proyecto es la variación de recursos para que los servicios puedan cumplir con el nivel de calidad acordado. La utilización de Marathon permite un caso de estudio que consiste en conseguir que las infinitas iteraciones del servicio no excedan una duración determinada.

Otro posible caso de estudio es que el servicio esté formado por la repetición de un mismo trabajo un número finito de veces y, además, el servicio deba terminar antes de un cierto tiempo. Con Chronos es posible implementar este caso de estudio, gracias a que ofrece la posibilidad de empezar a ejecutar un trabajo manualmente (mediante una llamada al

[10] Sitio web de Chronos: https://mesos.github.io/chronos/
[11] API REST de Chronos disponible en: https://mesos.github.io/chronos/docs/api.html

API REST o en la interfaz web) y, como se dice anteriormente, definir un número de iteraciones de un trabajo.

2.2.5. Monasca Openstack

Monasca[12] es una herramienta de código libre de monitorización (*Monitoring as a Service)* que se integra con OpenStack[13]. Esta solución es escalable, *multi-tenant* (un servidor varios agentes), tolerante a fallos y ofrece un rendimiento muy alto (según [6], puede procesar cientos de miles de métricas por segundo y almacenar los datos sin pérdidas mientras se procesan consultas).

Las métricas son la manera de almacenar la información de Monasca y están definidas por un nombre, el valor de la métrica y un diccionario de dimensiones. Cabe destacar que cada valor en un instante de tiempo se denomina medida o *measurement*. Por ejemplo, una medida de la métrica con nombre *cpu.system_time* sobre la utilización del tiempo de CPU del sistema dentro de un contenedor Docker *prueba1* en nodo *server1* es: *{"name": "cpu.system_time", "value": 190.0, "dimensions": { "container": "prueba1", "host": "server1"}}*.

El envío de métricas puede realizarse mediante un agente [14] (*monasca-agent*) o manualmente, utilizando la línea de comandos[15] *(command line interface o CLI)* o el API REST ([7]).

En Monasca, la autenticación se realiza mediante un *token* obtenido de OpenStack KeyStone[16]. Para ello, en cada proyecto de Monasca existen dos usuarios: uno para los agentes y otro genérico. El usuario genérico se utiliza para autenticarse en el CLI, el API REST o en otras herramientas (por ejemplo, la solución descrita en la sección 2.2.7) y así poder obtener el *token*.

El Agente de Monasca se instala en los nodos a monitorizar (por ejemplo, los agentes de Mesos) para obtener métricas. La monitorización con el agente se hace mediante la utilización complementos o *plugins*[17]. Monasca ofrece una serie de *plugins* para la monitorización del rendimiento del nodo y para los servicios y aplicaciones más utilizadas. Algunos ejemplos son: CPU, Memoria, Docker, Apache, Jenkins, Kubernetes, MySQL o SQL Server. Además, es posible definir *plugins* personalizados.

Aunque el método de monitorización suele ser mediante el Agente Monasca, es posible que en algún caso no sea posible su utilización. Por ello, el usuario puede tomar sus propias mediciones y, a continuación, enviarlas a Monasca mediante la línea de comandos o el API REST.

---

[12] Sitio web de Monasca: http://monasca.io/
[13] Sitio web de OpenStack: https://www.openstack.org/
[14] Documentación sobre monasca-agent: https://github.com/openstack/monasca-agent
[15] Documentación Monasca CLI: https://docs.openstack.org/cli-reference/monasca.html
[16] Sitio web de KeyStone: https://docs.openstack.org/developer/keystone/
[17] *Plugins* estándar del Agente Monasca: https://github.com/openstack/monasca-agent/blob/master/docs/Plugins.md

Una vez almacenadas las métricas, es posible realizar consultas (*statistics*) y crear alarmas. Las alarmas se componen de una expresión a evaluar y un método de notificación (*email*, *PagerDuty* o *Webhook*). Esta expresión se encarga de evaluar una o más métricas para determinar su estado. Según su estado las alarmas pueden clasificarse como deterministas o no deterministas. En el primer caso, son posibles dos estados: *OK* y *ALARM*. En el segundo caso, se añade el estado *UNDETERMINED*.

Las expresiones de las alarmas pueden estar compuestas por otras sub-expresiones, relacionadas con los operadores lógicos *and* y *or*. Cada expresión está formada por una función (*sum*, *avg*, *count*, *last*, *min* o *max*), una métrica (si es necesario, con las dimensiones), el periodo de la métrica (por defecto, 60 segundos), un operador relacional (*lt*, *gt*, *lte* o *gte*), el valor a comparar y el número de veces que debe ocurrir (por defecto, 1 vez).

```
<sub_expression>
    ::= <function> '(' <metric> [',' deterministic] [',' period] ')'
<relational_operator> threshold_value ['times' periods]
    | '(' expression ')'
```

Figura 6: Sintaxis de una expresión sin sub-expresiones dentro. Fuente: [7]

Si el lector desea profundizar más sobre la definición de métricas, medidas, alarmas, métodos de notificación o sobre el API REST, dispone de más información en [7].

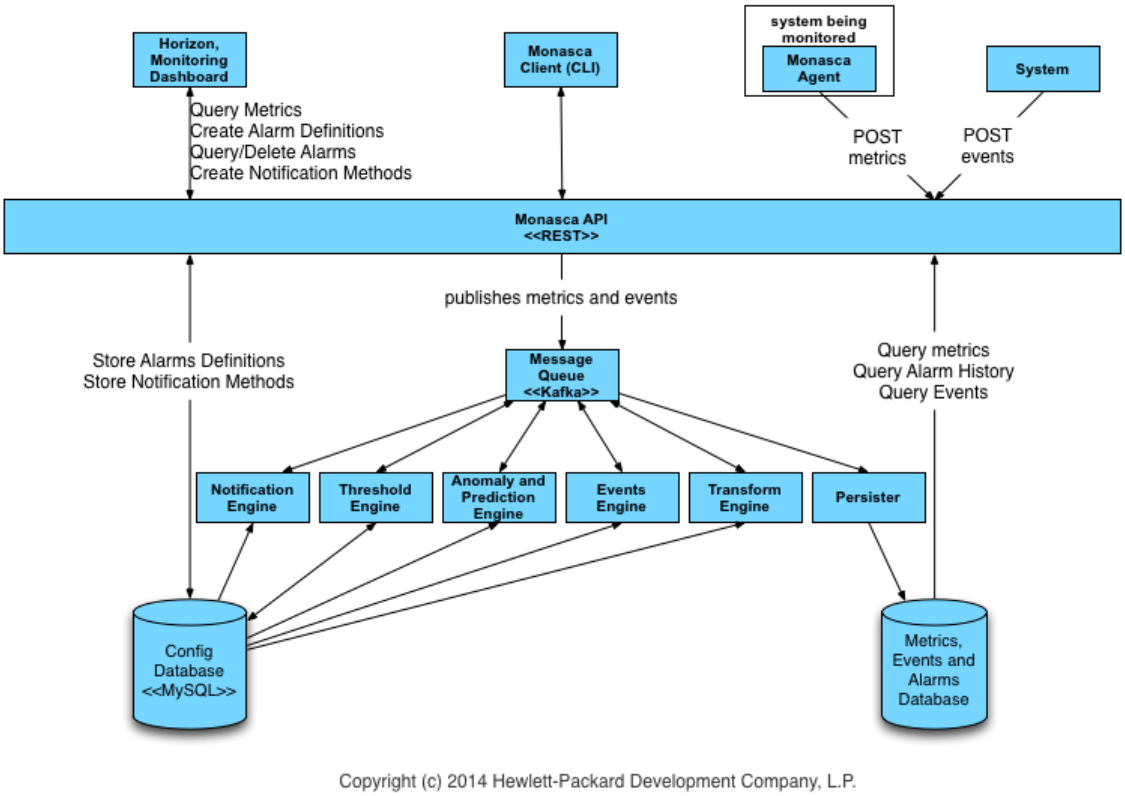

Figura 7: Arquitectura de Monasca OpenStack. Fuente: [6]

Según [6], la arquitectura de Monasca (ver Figura 7) está formada por los siguientes componentes:

- Agente (*monasca-agent*): se encarga de la monitorización mediante *plugins*.
- API de monitorización (*monasca-api*): REST API para la realización de consultas (estadísticas), envío de medidas, creación de métricas, alarmas y notificaciones.
- *Message Queue* (*MessageQ*)*:* recibe las métricas de *monasca-api* y los estados de las alarmas de *monasca-thresh* para que otros servicios puedan obtener esta información. Actualmente, se utiliza como cola de mensajes Apache Kafka[18].
- *Persister* (*monasca-persister*): obtiene métricas y estados de las alarmas de *MessageQ* y los almacena en la base de datos de Alarmas y Métricas.
- *Threshold Engine* (*monasca-thresh*): componente basado en Apache Storm[19] que evalúa las métricas y publica alarmas en *MessageQ*.
- *Notification Engine* (*monasca-notification*): obtiene los estados de las alarmas de *MessageQ* y envía, si es necesario, las notificaciones.
- *Analytics Engine* (*monasca-analytics*): obtiene los estados de las alarmas y las métricas de *MessageQ*. Después, detecta anomalías y busca correlaciones/agrupaciones de alarmas.
- Base de datos de Métricas y Alarmas: almacena las métricas y el historial de los estados de las alarmas. Actualmente, Monasca soporta Vertica y InfluxDB.

Aunque en la Figura 7 estén incluidos, hay algunos servicios que todavía no están disponibles:

- Motor de transformación y anexión (*monasca-transform*): transforma los nombres y algunos atributos de las métricas para crear nuevas métricas.
- Motor de predicción y anomalías: evalúa la predicción y las anomalías y predice métricas, así como probabilidades y calificaciones anómalas.

Después de un estudio sobre distintas herramientas de monitorización, se decidió utilizar Monasca debido a una serie de razones.

Una razón es la posibilidad de utilizar *plugins*, concretamente el que monitoriza contenedores Docker. La idea para cumplir con la calidad de servicio en Marathon era monitorizar el tiempo de CPU consumido en los contenedores Docker.

Otra razón es la posibilidad de enviar las métricas a través del API REST sin necesidad de tener instalada ninguna dependencia. Por ejemplo, en el caso de Chronos, se monitorizará el número de iteraciones restantes y el tiempo consumido.

Gracias a tener todas las métricas en el servidor, es posible realizar consultas sobre estadísticas mediante el API REST y, además, crear cuadros de mandos para ver el estado de las métricas (con la herramienta descrita en 2.2.7).

Otra posibilidad interesante es la creación de alarmas con las métricas. Sin embargo, debido a la limitación de que las sub-expresiones siempre evalúan una métrica y un valor numérico (no otra métrica), se decidió no utilizar esta característica.

[18] Sitio web de Apache Kafka: https://kafka.apache.org/
[19] Sitio web de Apache Storm: http://storm.apache.org/

2.2.6. CRIU

CRIU[20] es una herramienta software de Linux que permite realizar una copia del estado de una aplicación en ejecución como una colección de ficheros en disco (también llamado *checkpoint*). Gracias a estos ficheros, se puede restaurar la aplicación en el punto donde se realizó el *checkpoint*, incluso en otra máquina. Además, CRIU se implementa en el espacio del usuario y no en el núcleo.

Actualmente, CRIU está incluido dentro de OpenVZ[21], LXC[22] y Docker. Gracias a esta herramienta es posible realizar *checkpointing* en contenedores Docker (si el demonio de Docker se ejecuta bajo el modo "experimental").

2.2.7. Grafana Labs

Grafana Labs[23] es una herramienta de código abierto para la creación de cuadros de mandos o *dashboards* de información extraída de una gran cantidad de fuentes de datos ([8]). De manera nativa soporta Graphite, InfluxDB, OpenTSDB, Prometheus, Elasticsearch y AWS CloudWatch, aunque también soporta otras como Monasca o MySQL.

Se ha elegido Grafana como método de visualización de resultados porque puede obtener los datos de Monasca y permite crear *dashboards* muy complejos de una manera muy intuitiva. Un ejemplo de ello es la Figura 8, que representa el cuadro de mando utilizado en este proyecto utilizado para controlar las métricas de las aplicaciones de Marathon.

Figura 8: Cuadro de mando creado para visualizar datos de las aplicaciones ejecutadas utilizando Marathon.

[20] Sitio web de CRIU: https://criu.org/
[21] Sitio web de OpenVZ: https://openvz.org/Main_Page
[22] Sitio web de *Linux containers* (LXC): https://linuxcontainers.org
[23] Sitio web de Grafana Labs: https://grafana.com/

2.3 Entorno de desarrollo

2.3.1. Python

Python[24] es un lenguaje de programación de código abierto, alto nivel, interpretado, y multiplataforma.

Se ha elegido este lenguaje de programación debido a que es usual que esté instalada en los servidores y a la cantidad de librerías disponibles. Por ejemplo, se ha utilizado *Requests*[25] para la realización de peticiones a servicios REST, *Flask*[26] para la creación de un API REST y otras librerías del sistema como *threading*, *json*, *datetime*, *time*, *getopt* o *logging*.

2.3.2. REST

Según [9], *Representational State Transfer* o *REST* es una arquitectura que especifica restricciones, tales como una interfaz uniforme, que una vez aplicada a un servicio web produce un mejor rendimiento, escalabilidad y modificabilidad.

En la arquitectura REST, los datos y la funcionalidad son considerados recursos y son accesibles utilizando *Uniform Resource Identifiers* o *URIs*, los cuáles son normalmente enlaces web. Los recursos se usan mediante un conjunto de operaciones simples y bien definidas.

La arquitectura REST está compuesta por una arquitectura cliente-servidor y está diseñada para utilizar un protocolo de comunicación sin estado o *stateless (*los mensajes de petición son independientes), normalmente se utiliza HTTP. En esta arquitectura los clientes y servidores intercambian representaciones de los recursos mediante el uso de una interfaz y protocolo estandarizados.

En resumen, las aplicaciones se basan en los siguientes principios:

- Identificación de recursos a través de URIs: un servicio web *RESTful* expone un conjunto de recursos que identifican los objetivos de la interacción con sus clientes.
- Interfaz uniforme: los recursos se manipulan utilizando un conjunto fijo de cuatro operaciones de creación, lectura, actualización y eliminación: PUT, GET, POST y DELETE. PUT crea un nuevo recurso, que puede eliminarse mediante DELETE. GET recupera el estado actual de un recurso en alguna representación. POST transfiere un nuevo estado a un recurso.
- Mensajes auto-descriptivos: los recursos se desacoplan de su representación para que su contenido se pueda acceder en una variedad de formatos, como HTML,

[24] Sitio web de Python: https://www.python.org/
[25] Documentación del módulo *Requests* de Python: http://docs.python-requests.org/en/master/
[26] Documentación del módulo *Flask* de Python: http://flask.pocoo.org/

XML, texto sin formato, PDF, JPEG, JSON y otros. Los metadatos sobre el recurso están disponibles y se utilizan, por ejemplo, para controlar el almacenamiento en caché, detectar errores de transmisión, negociar el formato de representación adecuado y realizar la autenticación o el control de acceso.

- Interacciones de estado a través de hipervínculos: toda interacción con un recurso es sin estado. Existen varias técnicas de intercambio de estado, como URI reescritura, cookies, campos de formulario ocultos o también puede ir incrustado en el cuerpo del mensaje.

2.3.3. JSON

Según [10], *JavaScript Object Notation* o *JSON* es un formato ligero de intercambio de datos. Está formado por dos estructuras: una colección de pares de nombre/valor (también llamado diccionario, tabla hash o lista de claves) y una lista ordenada de valores (también llamada vector o lista).

En este proyecto se ha elegido *JSON* como formato estándar para el servicio REST desarrollado. La principal razón es que este formato es mucho más sencillo a la hora de codificar y decodificar la información, ya que basta con importar el módulo *json* de Python.

2.3.4. GitHub

Según [11], Git es un sistema de control de versiones distribuido de código abierto y gratuito. Git está diseñado tanto para gestionar proyectos pequeños como grandes con un gran rendimiento, facilidad y eficiencia.

Supera las herramientas de *Software Configuration Management* (SCM) como Subversion[27], CVS[28], Perforce[29] y ClearCase[30] con funciones como ramificación local, áreas de puesta en escena convenientes y múltiples flujos de trabajo.

En este proyecto se ha utilizado GitHub[31] como sistema de control de versiones, el cual está basado en Git, debido a que ofrece grandes funcionalidades como poder tener varias ramas o *branches*, es gratuito y es el repositorio oficial para los desarrollos del grupo de Grid y Computación de Altas Prestaciones de la Universitat Politècnica de València.

[27] Sitio web de Apache Subversion: https://subversion.apache.org/
[28] Sitio web de CVS: http://www.nongnu.org/cvs/
[29] Sitio web de Perforce: https://www.perforce.com/
[30] Sitio web de IBM ClearCase: http://www-03.ibm.com/software/products/es/clearcase
[31] Sitio web de GitHub: https://github.com/

Capítulo 3. Desarrollo del proyecto

El objetivo de esta sección es describir todo el trabajo realizado. La sección 3.1 comienza con descripción en líneas generales de los componentes y de la arquitectura. Las dos siguientes secciones describen detalladamente el comportamiento de los dos componentes que forman el sistema implementado: *Lanzador* y *Supervisor*. La sección 3.2 describe las limitaciones de la implementación, las modificaciones necesarias realizadas para la monitorización de los contenedores y cómo integrar una aplicación existente para su ejecución de acuerdo a factores de calidad de servicio, incluyendo una pequeña guía sobre cómo usar la implementación.

Para ayudar al lector a comprender mejor los componentes, se recuerda la funcionalidad del sistema para cada *framework*. En el caso de Chronos, el objetivo es ejecutar un trabajo un número determinado de veces antes de un cierto plazo de tiempo o *deadline*. En el caso de Marathon, el objetivo es que se garantice una determinada dedicación de tiempo de CPU a una tarea en un intervalo temporal. En ambos *frameworks* la aplicación debe ejecutarse dentro de un contenedor Docker.

Además, los componentes desarrollados en el trabajo se comunican con KeyStone, Monasca, Chronos y Marathon mediante un API REST utilizando módulos de Python implementados en este Trabajo Fin de Máster. El motivo por el cual se utilizan librerías propias es contener el número de dependencias en el proyecto (solo el módulo *Requests*). Estos módulos y todo el trabajo están disponibles en GitHub ([18]).

3.1. Diseño

3.1.1. Arquitectura del sistema

En esta sección se va a realizar un resumen en líneas generales sobre cómo funciona el sistema, en función del *framework*.

En Chronos, el componente *Lanzador* recibe un trabajo especificado en formato JSON. A continuación, envía información sobre este trabajo al componente *Supervisor* (a la ruta */initTask*), modifica el JSON para incluir las peticiones necesarias para monitorizar el trabajo y finalmente envía el trabajo.

A continuación, Chronos recibe y planifica el trabajo. Antes de terminar la primera iteración (una ejecución completa del *command* del JSON), se realiza la petición POST a *Supervisor* (petición a la ruta */updateTask*).

Cuando *Supervisor* recibe la petición de */updateTask*, se descarga la especificación del trabajo desde Chronos. Después, con los datos obtenidos de la petición */updateTask* y la

información obtenida en */initTask*, se calcula el estado de la aplicación, se actualizan los valores de *Supervisor* y se envían las métricas a Monasca.

En caso en que sea necesario la variación de los recursos asignados a la aplicación, se reajusta y se reenvía el trabajo a Chronos. Después de una cantidad de segundos (*sleeping_time*), *Supervisor* fuerza la ejecución de la siguiente iteración. La Figura 9 representa cómo interaccionan todos los componentes del sistema para aplicaciones Chronos.

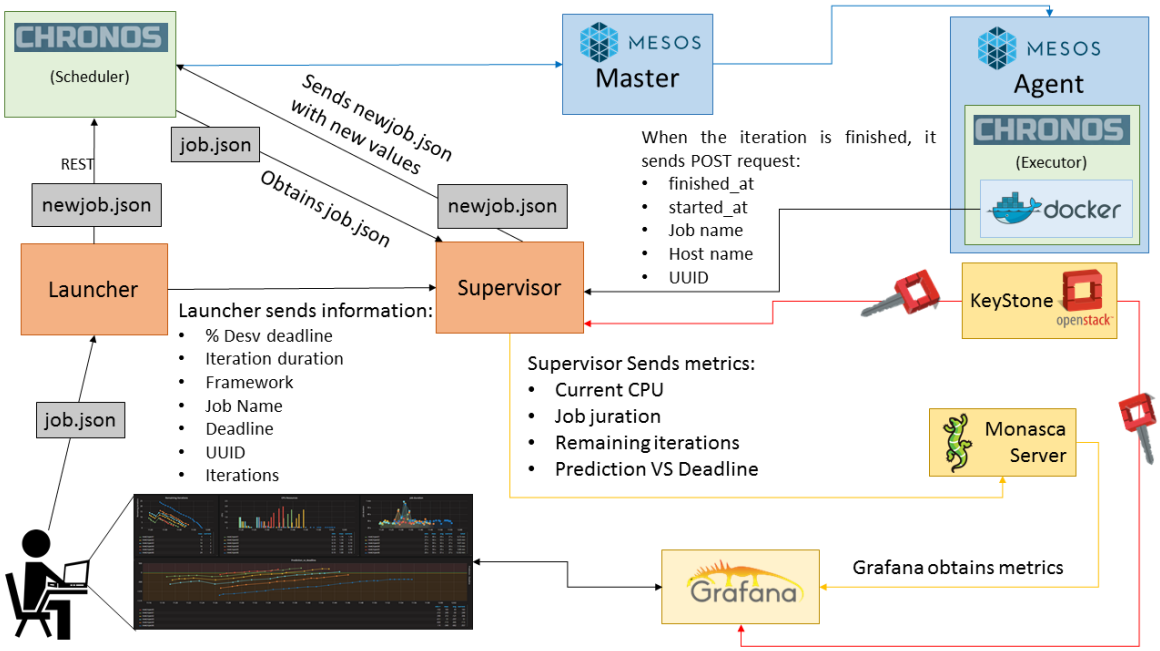

Figura 9: Arquitectura del sistema para aplicaciones que utilizan el framework Chronos.

Al igual que en Chronos, el componente *Lanzador* recibe un trabajo especificado en formato JSON. A continuación, envía información sobre este trabajo al componente *Supervisor* (a la ruta */initTask*).

En contraposición a Chronos, *Lanzador* no envía el trabajo modificado. En su lugar, envía un trabajo nuevo que ejecuta el programa *shell_trap*. Este programa se encarga de ejecutar el trabajo JSON en un contenedor Docker y de "avisar" a *Supervisor* del inicio y final de la ejecución (mediante las solicitudes a las direcciones */initMarathonApp* y */endMarathonApp*).

La monitorización se realiza mediante un agente Monasca que está instalando en todos los nodos. Este agente recoge métricas de todos los contenedores Docker que están ejecutándose en un determinado nodo y las envía al servidor Monasca.

A partir de las métricas del servidor Monasca, *Supervisor* comprueba el estado, se descarga la especificación del trabajo de Marathon, realiza la reasignación de recursos y lo vuelve a enviar.

Cuando la reasignación de recursos llegue a Marathon, este mandará una señal (*SIGTERM*) al trabajo que ejecuta *shell_trap*. Una vez ocurra esto, el programa la capturará dicha señal y realizará un *checkpoint*. La próxima vez que se inicie este programa ya se ejecutará con la nueva asignación de recursos. Al comienzo de una nueva ejecución, el programa detecta que hay un *checkpoint* disponible e inicia el contenedor Docker desde ese punto. Después, el programa se queda bloqueado (mediante *docker logs -f* container_name) hasta su finalización (o hasta que capture otra señal *SIGTERM*). Una vez finaliza el contenedor, *shell_trap* elimina el checkpoint.

La Figura 10 representa cómo interaccionan todos los componentes del sistema para aplicaciones Chronos.

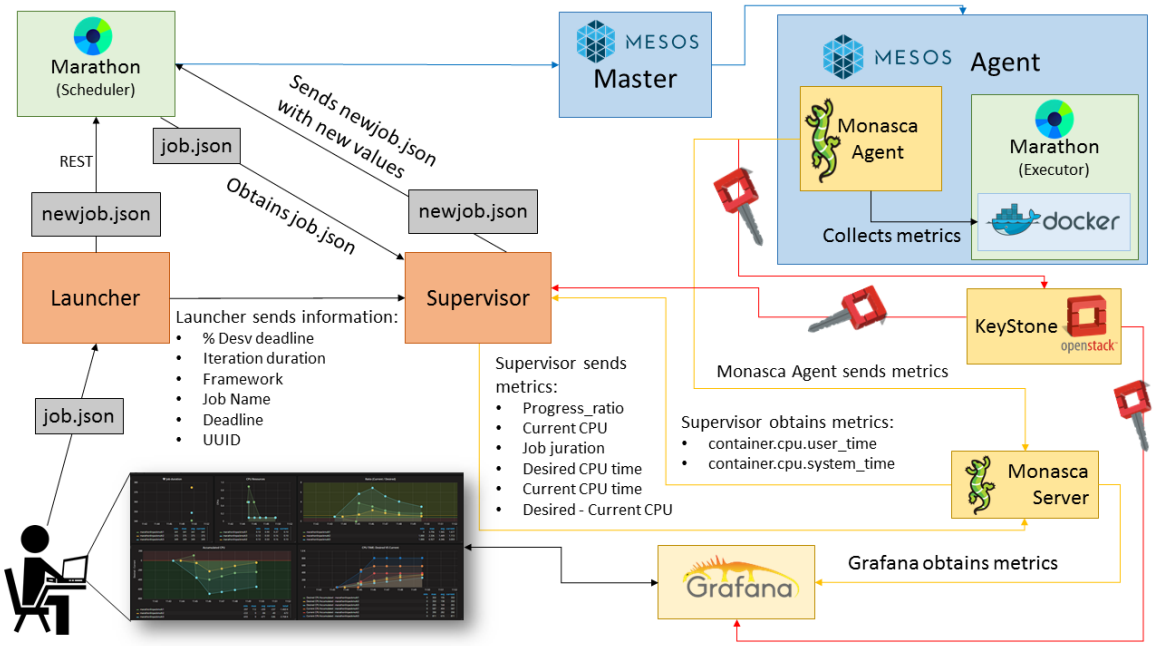

Figura 10: Arquitectura del sistema para aplicaciones que utilizan el framework Marathon.

3.1.2. Lanzador

El objetivo principal de este componente es iniciar la aplicación. Para ello, asigna un identificador único (*Universally unique identifier* o *UUID*) a cada trabajo, realiza los cambios necesarios en el JSON que especifica el trabajo, envía la información acerca del trabajo al *Supervisor* (se detallará este componente en la sección 3.1.3) y, finalmente, envía el trabajo al *framework* seleccionado.

I. Chronos

En primer lugar, se modifica el campo *command* del JSON que especifica el trabajo para hacer que el trabajo se repita indefinidamente (hasta que *Supervisor* lo detenga) e incluir una simple petición POST a la ruta */updateTask* de *Supervisor* donde envía el inicio y final de su ejecución.

El objetivo de incluir la petición POST es que el componente *Supervisor* pueda saber cuándo se ha realizado una iteración y cuánto ha durado para poder actuar antes de la siguiente iteración. Gracias a esta información, *Supervisor* sabe cuándo se han completado todas las iteraciones deseadas y puede eliminar el trabajo.

La razón por la cual el trabajo se repite indefinidamente es que, cuando el *scheduler* de Chronos planifica una tarea para un determinado instante de tiempo, puede ocurrir que en ese instante la tarea ya esté ejecutándose (porque *Supervisor* está forzando su inicio). Cuando se da este caso, el *scheduler* elimina la tarea planificada pudiendo ocasionar que no se ejecuten todas las iteraciones deseadas. Por este motivo, se repite el trabajo indefinidamente hasta que, gracias a la petición POST, *Supervisor* controla que se han completado todas las iteraciones y elimina el trabajo.

A continuación, se envía información en formato JSON a la ruta */initTask* del *Supervisor*. El motivo es que el componente *Supervisor* necesita información para poder realizar la monitorización y la elasticidad. Esta información está formada por el nombre del trabajo, el nombre del framework, la duración estimada de una iteración del trabajo, el UUID asociado, el plazo final en el que debe estar terminado todo el trabajo, el número de iteraciones que debe completar para que el trabajo se complete y el porcentaje de desviación del *deadline* que se utilizará para decidir si se deben disminuir los recursos asociados a este trabajo. Un ejemplo de JSON está disponible en la Figura 11.

Finalmente, se envía el trabajo Docker especificado en el JSON modificado a Chronos.

II. Marathon

Este componente envía la misma información al *Supervisor* que en el caso de Chronos para que este pueda identificar las peticiones realizadas por un determinado trabajo, monitorización y reajustar los recursos.

En contraposición con las aplicaciones Chronos, los trabajos ejecutados en Marathon no envían información al finalizar cada iteración, ya que se ejecutan de forma indefinida y se variarán sus recursos durante la misma ejecución.

Tal y como se ha expuesto en 2.2.3, Marathon permite realizar *checkpointing* de sus trabajos, pero solo cuando ocurre un fallo de su *scheduler* o del propio agente Mesos. Además, cuando se varían los recursos asociados a una aplicación en Marathon, el trabajo se replanifica y se ejecuta desde el inicio.

Atendiendo a este último detalle, no es posible aprovechar esta característica y es necesario que la implementación se encargue de realizar el *checkpoint* de la aplicación cuando se ocurra una reasignación de recursos. Dado que tampoco es posible indicar a Marathon que inicie un contenedor desde un punto de restauración, la única opción disponible es que la propia implementación se encargue totalmente de la gestión del contenedor Docker.

Por este motivo, este componente no envía un trabajo dentro de un contenedor Docker a Marathon, sino que envía una aplicación con la misma especificación de *hardware* que

ejecutará un programa que está en el espacio compartido de todos los nodos del clúster llamado *shell_trap*. Este programa se encargará de realizar las tareas restantes que se hacían en el otro *framework*: el envío del trabajo y monitorización.

Una tarea consiste en dar información sobre el inicio y final de la ejecución mediante peticiones POST al componente Supervisor. Al contrario que para los trabajos Chronos, en este caso se realizan dos envíos distintos para que *Supervisor* sea consciente de que el trabajo está en ejecución y pueda monitorizarlo.

Otra tarea consiste en la ejecución del trabajo dentro de un contenedor Docker, el cual puede ser un nuevo trabajo Docker o puede iniciarse desde un punto de restauración creado anteriormente. Por ello, es necesario que este programa realice una última tarea.

La última tarea consiste en la gestión de los *checkpoints* ya que este programa se encarga de crearlos y eliminarlos cuando es necesario. Cabe destacar que los *checkpoints* se realizan utilizando la herramienta de Docker y que se almacenan en un espacio compartido en todos los nodos del clúster.

Debido a cómo monitoriza el *plugin* Docker ([14]) de Monasca Agent los contenedores, es necesario guardar también el identificador del primer contenedor ejecutado en el espacio compartir en todos los nodos del clúster.

La creación del punto de restauración se realiza cuando se han variado los recursos asignados a la aplicación. Para ello se ha optado por capturar la señal *SIGTERM* ([13] la define como la señal genérica para ocasionar la finalización del programa) que es enviada por Marathon cuando se recibe una petición de cambio en la asignación de recursos.

El borrado de los puntos de restauración se realiza cuando la ejecución del contenedor Docker ha terminado. Para saber cuándo ha finalizado el contenedor, se mantiene el programa mediante *docker logs -f UUID*, comando que bloquea el programa hasta que finaliza el contenedor.

3.1.3. Supervisor

Este componente se encarga de realizar la monitorización de los trabajos y actuar en consecuencia. Para ello, implementa un servidor REST donde recibe distintas peticiones en función del *framework* que ejecuta el trabajo.

Además, mediante el módulo desarrollado para las interacciones con Monasca, recibe un *token* del servidor KeyStone para estar autorizado a enviar métricas y realizar consultas a Monasca.

La única petición en ambos *frameworks* es */initTask*. Esta petición es realizada por el componente *Lanzador* (descrito en la sección 3.1) que tiene un JSON como cuerpo del mensaje que contiene la información que puede visualizarse en la Figura 11.

Cabe recordar que, si esta petición corresponde a una aplicación Marathon, el campo *iterations* no existe.

```
{
	"uuid": "7986712f445b4902b0d6b857d4812440",
	"name": "linpackTest1",
	"desv_deadline": 0.2,
	"framework": "chronos",
	"deadline": 350,
	"job_duration": 35,
	"iterations": 12
}
```


Figura 11: Ejemplo JSON recibido de una petición /initTask para una aplicación Chronos.

I. Chronos

La monitorización de las aplicaciones Chronos se realiza cada vez que termina una iteración del trabajo. Como se ha descrito en la sección anterior, *Lanzador* incrusta en una petición POST a la ruta */updateTask* con un JSON que contiene: el UUID y el nombre de la aplicación, la máquina donde se ha ejecutado la iteración y el inicio y final de la iteración. Cabe destacar que tanto la máquina como el nombre del trabajo no son necesarias para la monitorización. Un ejemplo de JSON de esta petición puede verse en la Figura 12. A partir de este mensaje, el *Supervisor* obtiene la duración del trabajo, el host donde se ha ejecutado y actualiza el número de iteraciones restantes.

En el caso en el que no deba realizar más iteraciones, se elimina la aplicación de Chronos y de este componente.

En el caso contrario, utilizando la Ecuación 1 se calcula una predicción del instante de tiempo en el cual terminarán todas las iteraciones del trabajo.

$$prediction = time_{current} + remaining\_iterations * (job\_duration + sleeping\_time + deployment\_time)$$

Ecuación 1: Cálculo de la predicción del instante de tiempo en el que terminará el trabajo Chronos.

En esta expresión, *current_time* es el instante actual de tiempo expresado en *timestamp*. *sleeping_time* es el tiempo en segundos antes de forzar la siguiente iteración mediante el API REST de Chronos y *deployment_time* es el tiempo medio en segundos que tarda en desplegar un trabajo Chronos.


```
{
	"uuid": "7986712f445b4902b0d6b857d4812440",
	"name": "linpackTest1",
	"finished_at": 1497570924,
	"started_at": 1497570612,
	"hostname": "bigseawn7.localdomain"
}
```


Figura 12: Ejemplo JSON recibido de una petición /updateTask para una aplicación Chronos.

Una vez calculada la predicción, se obtiene el JSON del trabajo del API REST y se envían las métricas a Monasca con las que construir el cuadro de mando de Grafana sobre los trabajos Chronos.

Además de la predicción, se necesitan dos valores más para obtener el estado del trabajo: *deadline_superior y deadline_inferior*. El primero valor corresponde con el 95% del plazo máximo de finalización de la ejecución. El motivo de que sea un 95% es para poder tener un 5% del tiempo total más de tolerancia a error en la finalización del trabajo. El segundo valor corresponde al valor del *deadline* menos el porcentaje *desv_deadline* de la duración del trabajo. Por ejemplo, si el trabajo tiene una duración de *100* segundos y la desviación del *deadline* es *0.2*, el valor de *deadline_inferior* es *80*.

Con los tres valores ya calculados, se obtiene en qué estado se encuentra el trabajo. Si *prediction* es mayor que *deadline_superior*, entonces el estado es *UNDER_PERFORMANT*. Si *prediction* es menor que *deadline_inferior*, entonces el estado es *OVER_PERFORMANT*. En otro caso, el estado es ON_PLAN. En función del estado, se realiza un reajuste del valor de la CPU del trabajo.

Si ha sido necesario hacer el reajuste de los recursos, se envía el trabajo al *framework* para que, la próxima vez que se ejecute el trabajo, lo haga con la nueva asignación de recursos. Finalmente, se esperan *sleeping_time* segundos y se fuerza la ejecución del trabajo.

II. Marathon

Debido a que en las aplicaciones Marathon no hay un número de iteraciones determinado, la monitorización se basa en la comparación entre el tiempo de CPU consumido por el contenedor Docker que ejecuta el trabajo y el tiempo de CPU que debería haber consumido.

Cabe recordar que este contenedor Docker es el que se crea mediante el programa *shell_trap* (descrito en la subsección II de 3.1). Además, este programa realiza dos solicitudes POST al componente *Supervisor* para señalizar el inicio y final de la ejecución del trabajo, siendo las rutas */initMarathonApp* y */endMarathonApp*, respectivamente. La Figura 13 es un ejemplo de los cuerpos de los mensajes enviados en estas comunicaciones.

```
{
      "uuid": "7986712f445b4902b0d6b857d4812440",
      "started_at": "1497570612"
}
```

```
{
      "uuid": "7986712f445b4902b0d6b857d4812440",
      "finished_at": "1497570924"
}
```

Figura 13: Ejemplos de peticiones que realiza el programa cuando se inicia (imagen superior) y se termina (imagen inferior) la ejecución de una aplicación Marathon.

Para realizar esta monitorización se ha implementado una función llamada *main_marathon* que se ejecuta cada *monitoring_period* segundos. Esta función realiza que realiza una serie de tareas si la aplicación está ejecutándose, es decir, si se ha recibido una petición */initMarathonApp*.

La primera tarea de la función *main_marathon* es obtener el valor total del tiempo de CPU consumido (en segundos) por un contenedor: *cpu_current*. Para ello, se realizan dos consultas a Monasca para obtener *container.cpu.user_time* y *container.cpu.system_time* indicando como *name* el UUID asociado a la aplicación y como *start_time* el instante de tiempo que se inició el trabajo (ese instante es obtenido del cuerpo del mensaje de */initMarathonApp*). Ambas métricas son el total de *ticks de CPU* realizados por contenedor en el nodo donde se está ejecutando. Una vez obtenidas las métricas, se transforman a tiempo de CPU consumido en segundos dividiendo por los *ticks* por segundo del sistema (en la infraestructura donde se han ejecutado las pruebas este valor es 100).

Para que ambas métricas estén disponibles en Monasca, se han realizado varias acciones. Primero, se ha instalado el agente de Monasca en todos los nodos agentes del clúster Mesos. A continuación, se ha forzado que el nombre del contenedor asociado a una aplicación sea el UUID. Después, han configurado los agentes con un *plugin* para Docker ([14]), el cual monitoriza los contenedores Docker que estén ejecutándose en un determinado nodo. Para que la monitorización sea efectiva, ha sido necesario modificar el *plugin* Docker. Los motivos y las modificaciones se describen en la sección 3.1.4.

Después de obtener el valor real del tiempo de CPU consumido, es necesario calcular el valor deseado de CPU consumida en ese instante utilizando la Ecuación 2.

$$cpu\_desired = \begin{cases} \dfrac{(time_{current} - time_{init}) * job\_duration}{deadline} & \text{si } time_{current} \leq time_{init} + deadline \\ job\_duration & \text{si } time_{current} > time_{init} + deadline \end{cases}$$

Ecuación 2: Cálculo del valor de CPU consumida que debería llevar un contenedor en un instante de tiempo.

Una vez se poseen los valores de *cpu_current* y *cpu_desired*, se puede calcular el porcentaje de progreso (*progress_ratio*) en un instante de tiempo utilizando la Ecuación 3.

$$progress\_ratio(t) = \frac{cpu\_current(t)}{cpu\_desired(t)}$$

Ecuación 3: Cálculo del porcentaje de progreso de la ejecución en un instante t

Una vez calculados todos los valores, se envían las métricas deseadas a Monasca para que puedan visualizarse en el *dashboard* construido para Marathon. Un ejemplo de este tipo de *dashboard* está disponible en la Figura 8.

Utilizando los porcentajes desv_*deadline* y *progress_ratio*, se calcula en qué estado se encuentra el trabajo. Si *progress_ratio* es inferior a *(1.0-desv_deadline)*, entonces el estado es *UNDER_PERFORMANT*. Si *progress_ratio* es superior a

(1.0+desv_deadline), entonces el estado es *OVER_PERFORMANT*. En otro caso, el estado es *ON_PLAN*.

Finalmente, si el estado no es ON_PLAN se reajusta el valor de la CPU y se envía al *framework*. A partir de este momento, Mesos enviará la señal *SIGTERM* que capturará el programa *shell_trap* para poder crear el punto de restauración antes de que Mesos elimine el trabajo para realizar el reasignamiento de recursos.

3.1.4. *Plugin* Docker para el agente de Monasca

El *plugin* Docker realiza la monitorización utilizando los pseudo-ficheros ubicados en en */sys/fs/cgroup/*. En este trabajo, son necesarios los valores obtenidos del fichero *cpuacct.stat* ubicado en */sys/fs/cgroup/cpuacct/docker/$CONTAINER_ID/*, donde *$CONTAINER_ID* es el identificador único asignado por el demonio Docker al contenedor a monitorizar.

```
$ cat /sys/fs/cgroup/cpuacct/docker/$CONTAINER_ID/cpuacct.stat
> user 2451  # time spent running processes since boot
> system 966 # time spent executing system calls since boot
```

Figura 14: Fichero utilizado que contiene los ticks consumidos por el contenedor con identificador $CONTAINER_ID. Fuente:[17]

Este plugin se utiliza para monitorizar los contenedores utilizados en las aplicaciones Marathon y realizar la reasignación de recursos. Estos contenedores pueden ser una nueva ejecución del trabajo o un contenedor auxiliar (creado pero no iniciado), al cual se le indicará el *checkpoint* desde donde iniciarse.

Cuando un contenedor es una nueva ejecución, Docker le asigna un identificador y, utilizando ese identificador como $CONTAINER_ID para acceder al fichero de la Figura 14, el *plugin* Docker monitoriza la CPU consumida (entre otros parámetros).

Cuando el contenedor se crea para iniciar desde un punto de restauración, Docker también le asigna un identificador. Sin embargo, la CPU consumida no puede obtenerse del fichero anterior si se utiliza ese identificador como $CONTAINER_ID. Para poder obtener esos valores, es necesario que $CONTAINER_ID sea el identificador asignado por demonio Docker al contenedor al que se le realizó el primer punto de restauración, es decir, al contenedor que inició una nueva ejecución.

Este *plugin* ha tenido que modificarse debido a que solo monitoriza los contenedores que están ejecutándose en esa máquina, es decir, solo monitoriza los contenedores cuyo identificador haya sido creado por el demonio Docker en esa misma máquina. La solución a este problema se consigue mediante dos acciones.

Primero, cuando *shell_trap* inicia un contenedor con identificador *$CONTAINER_ID* desde un *checkpoint*, crea un enlace simbólico del fichero */sys/fs/cgroup/cpuacct/docker/$STAT_CONTAINER_ID/cpuacct.stat* en */opt/mycgroup/cpuacct/$CONTAINER_ID/* donde *$STAT_CONTAINER*_ID es el identificador del contenedor del cual se realizó el primer *checkpoint*.

Después, debe modificarse la función *_report_cgroup_cpuacct* del código fuente del plugin para que utilice este pseudo-fichero en lugar del obtenido con el identificador */$CONTAINER_ID*. Las modificaciones pueden verse en la Figura 15.

```
107     def _report_cgroup_cpuacct(self, container_id, container_dimensions):
108         stat_file = self._get_cgroup_file('cpuacct', container_id, 'cpuacct.stat')
109         # Edit by serlophug
110         dirpath = '/opt/mycgroup/cpuacct/' + container_id
111         if os.path.isdir(dirpath):
112             stat_file = dirpath + '/cpuacct.stat'
113         # End edit
114         stats = self._parse_cgroup_pairs(stat_file)
115         self._report_rate_gauge_metric('container.cpu.user_time', stats['user'], container_dimensions)
116         self._report_rate_gauge_metric('container.cpu.system_time', stats['system'], container_dimensions)
```

Figura 15: Función modificada del plugin Docker del agente de Monasca. Fuente: [18]

3.2. Implementación

3.2.1. Restricciones

La utilización de esta implementación conlleva una serie de restricciones sobre qué tipo de aplicaciones se puede utilizar.

Una primera restricción es que solo es posible ejecutar trabajos dentro de contenedores Docker. Sin embargo, cabe destacar que este proyecto surge con el objetivo de variar los recursos de contenedores Docker para cumplir con el nivel de calidad, por lo que esto no es realmente una limitación de la implementación.

En el caso de Marathon es posible definir aplicaciones que se estén ejecutando en varias instancias a la vez. Esta implementación asume que los trabajos solo utilizan una máquina.

Las mayores restricciones de esta implementación están impuestas por la utilización de *checkpointing*.

Además, los agentes de Mesos deben disponer de un directorio compartido donde se almacenarán los *checkpoints* y el programa *shell_trap*. No obstante, esta limitación podría eliminarse fácilmente implementando un sistema de distribución de los *checkpoints*.

3.2.2. ¿Cómo adaptar una aplicación?

Para que una aplicación sea compatible con esta implementación debe cumplir con las restricciones descritas en la sección anterior, el contenedor Docker debe tener instalado el paquete *curl* [32]y, en el JSON donde se especifica el trabajo, debe incluir un campo que tenga como clave *qos* y como valor un diccionario (un ejemplo está disponible en Figura 16). Este diccionario tiene tres elementos: *duration*, *desv_deadline* y *deadline*.

El elemento *duration* corresponde con una aproximación de la cantidad de tiempo consumida por la CPU de una ejecución (en Chronos, esta cantidad corresponde a una sola iteración). El valor de este elemento debe ser segundos.

En el caso de Chronos, el elemento *deadline* se refiere al plazo máximo para que terminen todas las iteraciones. En el caso de Marathon, el elemento *deadline* se refiere a la duración máxima de una ejecución. El valor de este elemento debe ser segundos.

El elemento *desv_deadline* es el tanto por ciento que se utiliza en la comprobación del estado de un trabajo. En el caso de Chronos, es el porcentaje máximo que comprueba cuanta diferencia existe entre la predicción y el *deadline*. En el caso de Marathon, este porcentaje se utiliza para ver cuánto se desvía el porcentaje de progreso de la perfección (1.0). El valor de este elemento debe expresado como resultado de una fracción.

```
{
      ...
      "qos": {
            "duration": 600,
            "desv_deadline": 0.2,
            "deadline": 350
            }
      ...
}
```

Figura 16: Ejemplo de especificación del nivel de calidad para una aplicación.

En el caso en que la aplicación vaya a ejecutarse en Marathon, es necesario que la aplicación soporte la creación y restauración de los *checkpoints* mediante Docker.

3.2.3. Utilización de los componentes

Como se dice en 3.2.1, tanto KeyStone como los *frameworks* utilizan usuario y contraseña como medio de autenticación. Cabe recordar que con la obtención de un *token* mediante una petición al servidor KeyStone, la implementación podrá autenticarse en Monasca.

[32] Sitio web de *curl*: https://curl.haxx.se/

Debido a que pasar como argumentos todos los parámetros necesarios para autenticarse sería tedioso, se ha decidido obtener las credenciales a través de ficheros con formato JSON pasados como argumentos.

En el caso de los *frameworks* es necesario un fichero como el representado en la Figura 17 (los puertos por defecto de Chronos y Marathon son 4400 y 8080).

```
{
      "url": "http://IP:PORT",
      "user": "USER",
      "password": "PASSW"
}
```

Figura 17: JSON que define las credenciales de Chronos y Marathon.

Para poder conectarse a Monasca es necesario disponer de las credenciales para KeyStone y la dirección web donde realizar las peticiones a Monasca. Para ello, es necesario un fichero con los mismos campos que la Figura 18 (los puertos por defecto de KeyStone y Monasca son 35357 y 8070).

```
{
      "keystone_url": "http://IP:PORT",
      "monocasca_client_url": "http://IP:PORT",
      "username": "USER",
      "password": "PASSW",
      “project_name”: "myprojectname"
}
```

Figura 18: JSON que define las credenciales de KeyStone y Monasca.

El componente *Lanzador* tiene los siguientes argumentos:

- *-j <job-file>*. Parámetro obligatorio. *<job-file>* es el fichero donde se encuentra la especificación del trabajo Docker en formato JSON. Debe contener la información sobre calidad de servicio (Figura 16).
- *-m <credentials-Marathon>*. Parámetro obligatorio si no se utiliza -c. *<credentials-Marathon>* es el fichero donde se encuentra las credenciales de Marathon. Debe ser como el de la Figura 17.
- *-c <credentials-Chronos>*. Parámetro obligatorio si no se utiliza -m. *<credentials-Chronos>* es el fichero donde se encuentra las credenciales de Chronos. Debe ser como el de la Figura 17.
- *-i <supervisor-url>*. Parámetro obligatorio. *<supervisor-url>*. Es la dirección web del directorio raíz del API REST del comoponente *Supervisor*. Por ejemplo: “http://10.0.0.2:30000”.

Además de estos parámetros, el usuario puede cambiar algunas variables de configuración dentro de la función *main de* este componente:

- *shell_trap_path*: cadena de texto con la ruta absoluta del programa *shell_trap*.
- *checkpoint_dir*: cadena de texto con la ruta absoluta del directorio donde se almacenarán los puntos de restauración.

- *taskKillGracePeriodSeconds*: entero que permite cambiar el tiempo que tiene un trabajo Marathon cuando ha recibido la señal de eliminación.

El componente *Supervisor* tiene los siguientes argumentos:

- *-m* <*credentials-Marathon*>. Parámetro obligatorio. <*credentials-Marathon*> es el fichero donde se encuentra las credenciales de Marathon. Debe ser como el de la Figura 17.
- *-c* <*credentials-Chronos*>. Parámetro obligatorio. <*credentials-Chronos*> es el fichero donde se encuentra las credenciales de Chronos. Debe ser como el de la Figura 17.
- *-o* <*credentials-Monasca*>. Parámetro obligatorio. <*credentials-Monasca*> es el fichero donde se encuentra las credenciales de Marathon. Debe ser como el de la Figura 18.
- *-i* <*rest-ip*>. Parámetro opcional (valor por defecto: "0.0.0.0").
- *-p*<*rest-port*>. Parámetro opcional (valor por defecto: "30000").

Al igual que en el caso de *Lanzador*, este componente tiene otros parámetros de configuración configurables desde el fichero:

- *logDirectory*: cadena de texto con la ruta absoluta del directorio donde se almacenarán los ficheros de *logs*.
- *cpu_max_slave*: entero que define el valor de CPU máximo en los nodos del clúster.
- *cpu_max_slave*: entero que define el valor de CPU mínimo en los nodos del clúster.
- *cpu_increment*: entero que define la cantidad de CPU que aumenta cuando la aplicación está en estado UNDER_PERFORMANT.
- *cpu_decrement*: entero que define la cantidad de CPU que disminuye cuando la aplicación está en estado OVER_PERFORMANT.
- *monitoring_period*: entero que define cada cuantos segundos se comprueba el estado de las aplicaciones Marathon.

Capítulo 4. Casos de estudio

En este capítulo se describe tanto la aplicación utilizada como los casos de estudio diseñados para probar el sistema desarrollado en este Trabajo Final de Máster.

4.1. Aplicación seleccionada

Debido a que el sistema implementado se encarga de variar la CPU asignada a cada trabajo, se ha elegido como trabajo para los distintos casos de estudio el *benchmark* LINPACK ([15]). Se ha utilizado una versión editada del *benchmark* disponible en [16]. Los cambios realizados en esta versión son la posibilidad de cambiar el tamaño de la matriz, la eliminación de las pruebas secuenciales y se ha paralelizado con OpenMP[33] todo el programa (antes solo lo estaba la función *msgefa*).

A continuación, se ha compilado el programa incluyendo todas las librerías en el ejecutable, encapsulado dentro de una imagen Docker y se ha almacenado en Docker Hub. Tanto el código fuente como el ejecutable están disponibles dentro de la imagen Docker[34] generada.

4.2. Chronos

Los casos de estudio de Chronos consisten en ejecutar un trabajo un número determinado de veces antes de un cierto plazo de tiempo o *deadline*. La definición del plazo de tiempo debe ser realista ya que debe hacerse considerando tanto el tiempo de despliegue entre iteraciones (unos 45 segundos) como el tiempo medio de ejecución con una CPU asignada.

Como se ha descrito en las secciones anteriores, el sistema realiza una predicción lineal para saber si podrá cumplir con el nivel de calidad acordado. Por este motivo es interesante saber cómo se comporta el sistema tanto con una evolución lineal como no lineal.

El caso I tiene como objetivo que la evolución del progreso del trabajo aumente de manera lineal. Por este motivo, la duración de los trabajos debe ser muy similar. Para realizar el estudio del caso I se han definido seis trabajos: *openmpLinpack1*, *openmpLinpack2*, *openmpLinpack3*, *openmpLinpack4*, *openmpLinpack5* y *openmpLinpack6*. Estos trabajos tienen objetivo ejecutar el test LINPACK con un tamaño de matriz (el mismo en todas las iteraciones) y número determinado de iteraciones antes del fin de plazo. Concretamente, el *deadline* para todos los trabajos es media hora (mil ochocientos segundos). Este plazo debe especificarse de manera realista,

[33] Sitio web OpenMP: http://www.openmp.org/
[34] Imagen Docker utilizada para las pruebas: https://hub.docker.com/r/serlophug/linpacktest/

considerando que existe un plazo de despliegue en cada iteración de alrededor de treinta segundos y de la diferencia que existe entre ejecutar con una o varias CPU.

La Tabla 1 describe los trabajos del caso I. Cabe destacar que la duración especificada en el apartado de QoS es diferente a la duración con una CPU porque, si el trabajo a ejecutar lo permite, Docker lo ejecuta con todas las CPU que tengan disponible en ese instante (en la infraestructura utilizada en este trabajo, el número máximo de CPUs es dos).

Nombre del trabajo	Iteraciones	QoS - Duración (s)	Tamaño matriz	Duración con una CPU (s)	Duración con dos CPU (s)
openmpLinpack1	4	400	6000	480.4	362.2
openmpLinpack2	4	400	6000	480.4	362.2
openmpLinpack3	13	130	4000	162.75	113.2
openmpLinpack4	11	130	4000	162.75	113.2
openmpLinpack5	12	130	4000	162.75	113.2
openmpLinpack6	14	80	3500	92.36	66.32

Tabla 1: Información sobre los trabajos del caso I.

El objetivo del caso II es estudiar cómo se comporta el sistema cuando el tiempo de ejecución del trabajo no crece de forma lineal. Para ello, se redefinieron los trabajos anteriores de manera para que ejecuten el test LINPACK con el mismo número de iteraciones, pero realizando una pequeña variación del tamaño de la matriz (sumando o restando 250 unidades de manera aleatoria) en cada iteración.

```
{

        "schedule": "R4//PT5M",
        "name": "openmpLinpack1",
        "container": {
           "type": "DOCKER",
           "image": "serlophug/openmplinpack",
           "forcePullImage": true,
           "volumes":
               [ {
                     "containerPath": "/shared/",
                     "hostPath": "/home/users/shared/serlophug/",
                     "mode": "RW"
               } ]
         },
         "cpus": "1",
         "mem": "500",
         "command": "/linpack_openmp 6000",
         "owner": "serlophug",
         "async": "false",
         "qos":
            {
              "duration": 400,
              "desv_deadline": 0.2,
              "deadline": 1800
            }
}
```

Figura 19: Ejemplo de especificación de un trabajo Chronos utilizado en el sistema. Concretamente, esta especificación corresponde a la aplicación openmLinpack1 del caso de estudio I.

4.3. Marathon

El caso de estudio de Marathon consiste en garantizar una determinada dedicación de tiempo de CPU a una tarea durante un intervalo temporal realizando una variación de la asignación de recursos y utilizando puntos de restauración de Docker.

El caso de estudio III corresponde con la aplicación *openmpLinpack1* especificada en la Tabla 1, pero realizando la variación durante el tiempo de ejecución, sin un número determinado de iteraciones y con una desviación del *deadline* de 0.1.

La aplicación *openmpLinpack1* tiene un coste temporal medio, en función de las CPUs que utilice, entre 362 y 480 segundos. Por este motivo, se ha definido una duración media de 400 segundos.

El objetivo de este caso de estudio es ofrecer una calidad de servicio que garantice que la aplicación se completará siempre en un intervalo temporal menor que la duración media de ejecutar la aplicación con una CPU.

```
{
      "id": "/openmplinpack1",
      "cmd": "/linpack_openmp 6000",
      "instances": 1,
      "cpus": 1,
      "mem": 512,
      "disk": 0,
      "container": {
           "type": "DOCKER",
           "docker": {
             "network": "HOST",
             "image": "serlophug/linpacktest",
             "forcePullImage": true
            }
       },
       "backoffSeconds": 300,
       "qos": {
           "duration": 400,
           "desv_deadline": 0.1,
           "deadline": 480
       }
}
```

Figura 20: Ejemplo de especificación de trabajo Marathon utilizado en el sistema. Concretamente, esta especificación corresponde a la aplicación openmLinpack1 del caso de estudio III.

Capítulo 5. Resultados y discusión

En este capítulo se realiza un estudio de los resultados obtenidos en los casos de estudio descritos en el Capítulo 4.

5.1. Caso I:

Este caso de estudio consiste en estudiar cómo se comporta el sistema cuando el tiempo de ejecución del trabajo crece de forma lineal, es decir, todas las iteraciones tienen la misma duración.

Cabe destacar que es imposible que todas las iteraciones con los mismos recursos asignados tengan la misma duración, pero se asumirá que sí se cumple esta propiedad. La razón de que no se cumpla esta igualdad es que Docker utiliza toda la CPU disponible en el nodo donde se está ejecutando.

Los resultados obtenidos pueden visualizarse en la Figura 21. Esta figura representa la diferencia entre la predicción calculada utilizando la Ecuación 1 y el instante temporal en el que deben haber terminado todas las iteraciones de la aplicación. El valor resultante permite obtener el estado de la aplicación respecto al *deadline*:

- Si el valor se encuentra en la región roja, el rendimiento es inferior al deseado y es necesario aumentar los recursos asignados a esa aplicación.
- Si el valor se encuentra en la región amarilla, el rendimiento es inferior al deseado y es necesario disminuir los recursos asignados a esa aplicación.
- Si el valor se encuentra en la región verde, el rendimiento es superior al esperado, pero no permite completar todas las iteraciones antes del 95% del tiempo deseado.
- Si el valor es igual a cero (línea blanca), significa que la predicción calculada prevé que se completarán todas las iteraciones en el instante justo del plazo, es decir, la ejecución se habrá ajustado de manera perfecta.

Las Figuras 22-27 representan la CPU asignada y la duración de cada iteración. Dado que la infraestructura donde se han realizado las pruebas disponía de cinco nodos con dos CPU cada uno, en algunos casos puede verse la correlación entre la CPU asignada y la duración ya que no se disponía de más recursos que de los que las aplicaciones tenían asignados.

La razón de que no estén correlacionados todos los casos (por ejemplo, la Figura 27) es que Docker utiliza toda la CPU disponible en el nodo, aunque sea superior a la asignada.

Pese a que no exista una correlación entre la CPU y la duración de las iteraciones, sí que existe una correlación inversamente proporcional entre los valores que representan la diferencia entre la predicción y el *deadline* de cada aplicación en la Figura 21 y la CPU asignada a cada aplicación en las Figuras 22 - 27 .

Gracias al sistema desarrollado en este trabajo, cuatro de los seis trabajos han acabado en un tiempo inferior al plazo dado, siendo este plazo inferior a la duración del trabajo con una sola CPU asignada.

Dado que la duración media esperada era inferior a la duración utilizando una CPU, el sistema ha estado congestionado y esto ha propiciado que fuera más complicado que todas las aplicaciones terminaran a tiempo. Además, cabe destacar que los dos trabajos que no han terminado a tiempo, *openmpLinpack3* y *openmpLinpack5*, han tenido una iteración con una duración muy superior incluso al tiempo medio de una sola CPU (más de un 120% de duración).

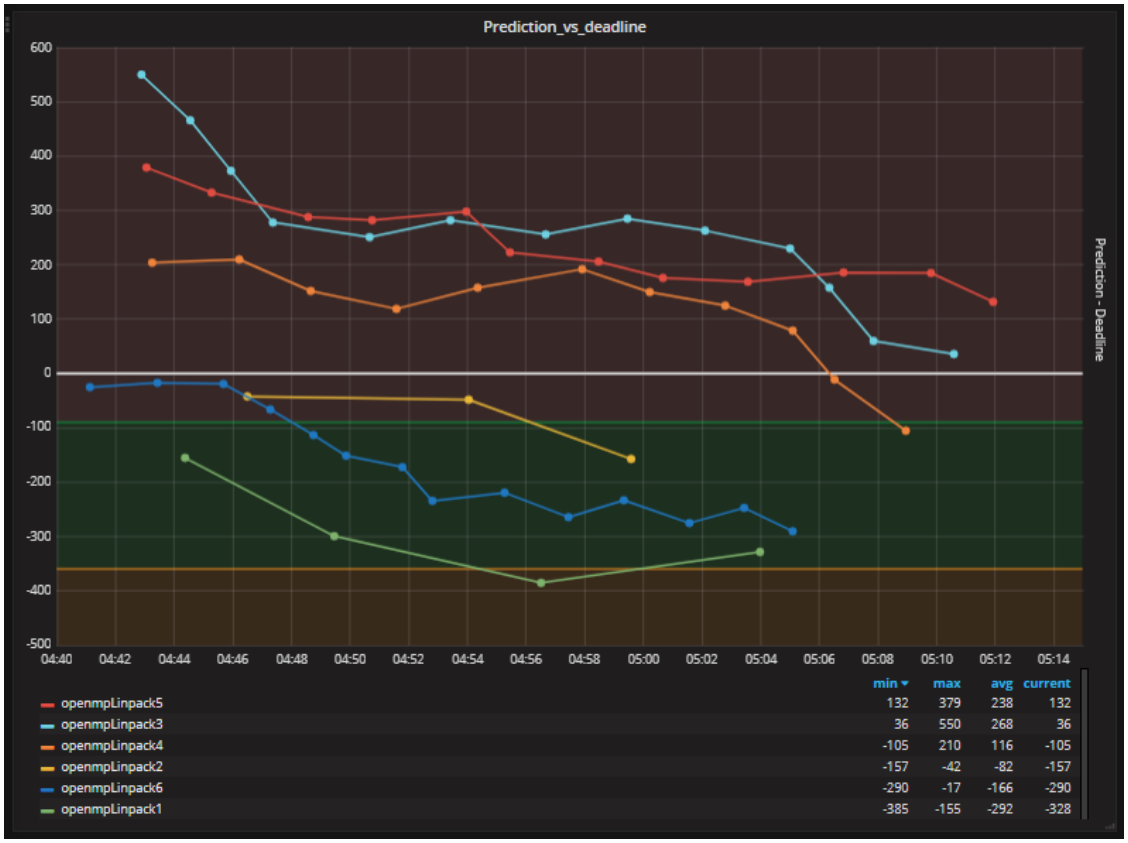

Figura 21: Diferencia entre la predicción calculada y el plazo que debe cumplirse en el caso I.

Figura 22: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack1.

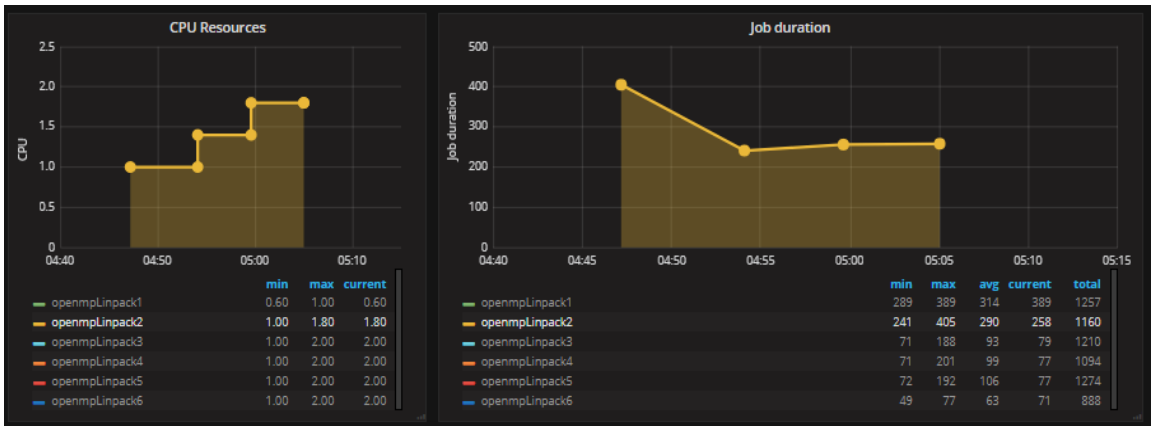

Figura 23: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack2.

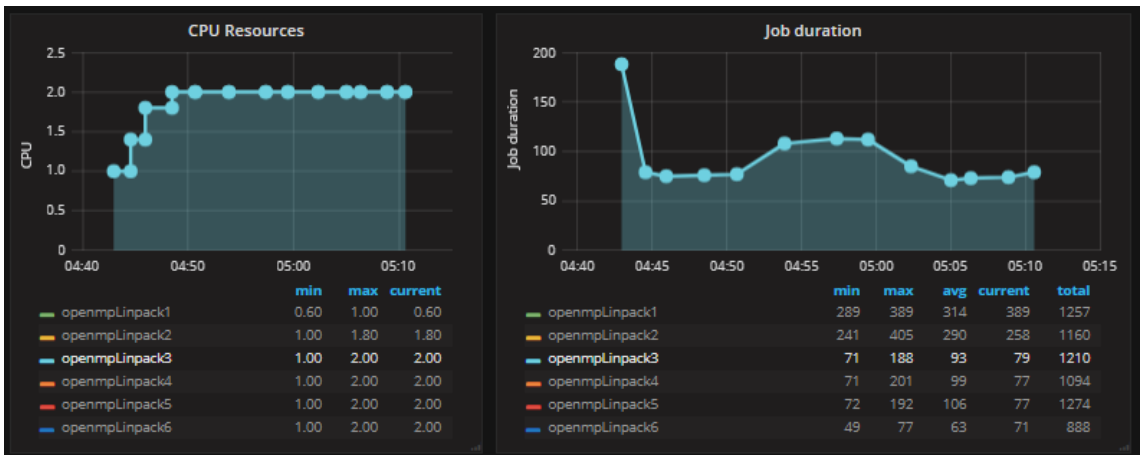

Figura 24: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack3.

Figura 25: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack4.

Figura 26: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack5.

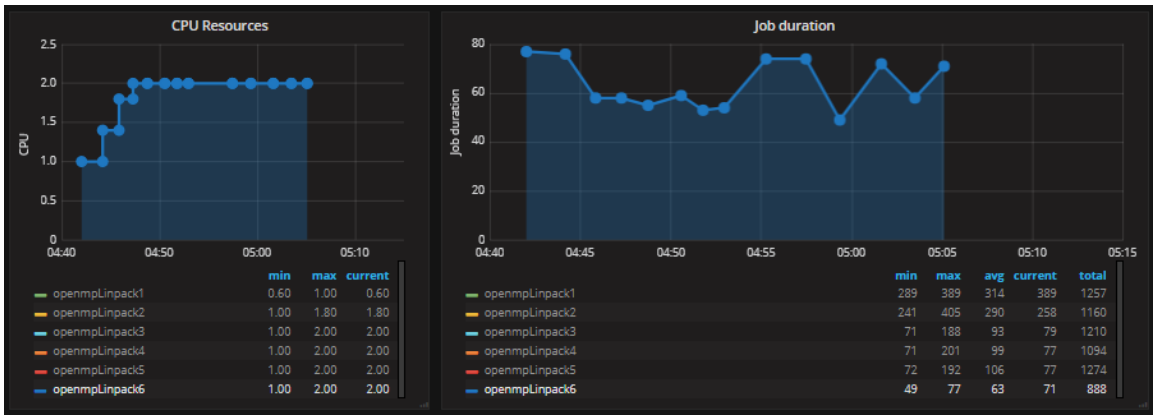

Figura 27: Valor de CPU asignado en cada iteración y duración de cada iteración de la aplicación openmLinpack6.

5.2. Caso II

Este caso de estudio consiste en analizar cómo se comporta el sistema cuando el tiempo de ejecución del trabajo crece de forma no lineal. Para ello, se ha variado de forma aleatoria en 250 el tamaño de la matriz en cada iteración de los trabajos utilizados en el caso I.

Al realizar la variación de forma aleatoria, el objetivo de que la evolución de las iteraciones haya sido no lineal se ha cumplido y esto ha propiciado que el comportamiento del sistema varíe en función de la aleatoriedad de los valores del tamaño de las matrices.

La Figura 28 representa una muestra de este caso de estudio, donde el sistema ha podido cumplir con la calidad de servicio en cinco de las seis aplicaciones.

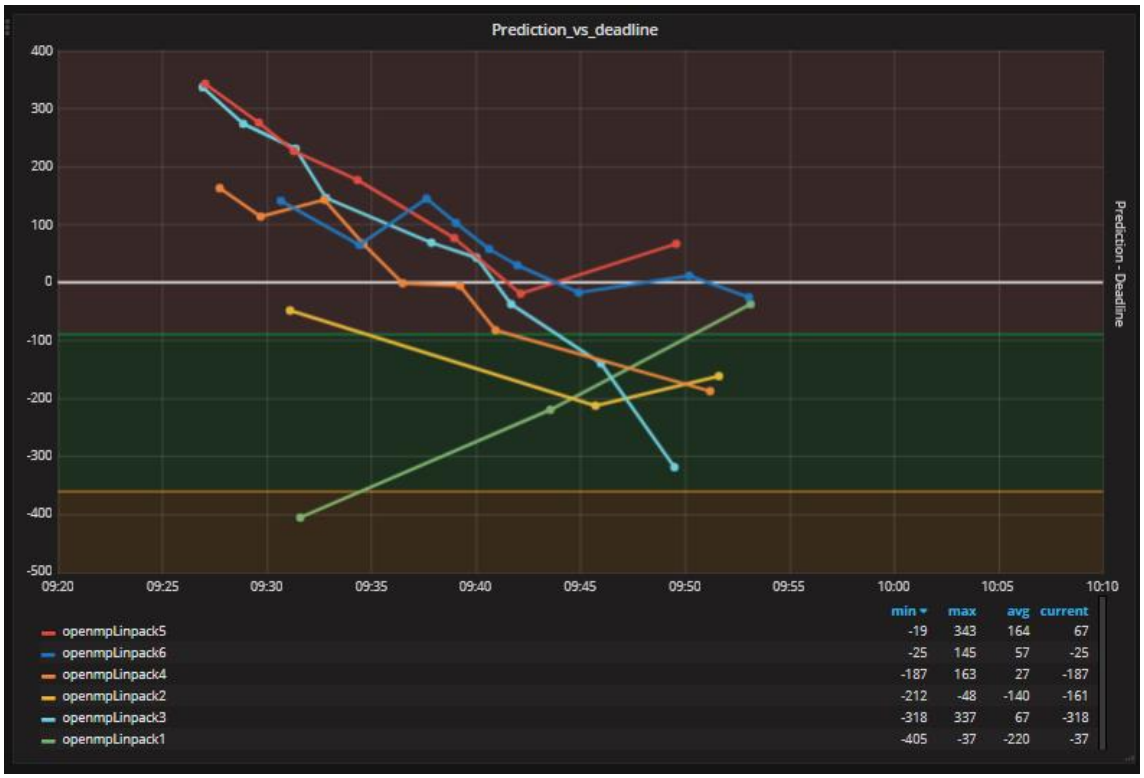

Figura 28: Diferencia entre la predicción calculada y el plazo que debe cumplirse en el caso II.

5.3. Caso III

Este caso de estudio consiste en garantizar que la ejecución de la aplicación *openmpLinpack1* se completa en un tiempo menor a la duración media de las ejecuciones utilizando una sola CPU. Gracias a la utilización del sistema se ha conseguido ofrecer la calidad de servicio deseada, consiguiendo que la aplicación se completara con 87 segundos de margen, es decir, en 393 segundos.

Para alcanzar esta cifra, el sistema ha variado la CPU asignada a la aplicación dos veces, como puede visualizarse en la Figura 31. El reajuste de CPU se realiza considerando la tasa de progreso calculada mediante la Ecuación 3 en cada instante temporal, representado en la Figura 29. La Figura 30 permite visualizar como varían los valores que se utilizan para calcular la tasa de progreso: tiempo consumido real y deseado de CPU.

A continuación, va a describirse el transcurso de la ejecución. Primero, se ha ejecutado el trabajo consiguiendo un rendimiento superior al esperado. Por este motivo, en el instante 2:33:30, se puede ver en Figura 30 como el tiempo de CPU consumido supera al tiempo de CPU a consumir. Como consecuencia, el valor representado correspondiente a ese instante de tiempo en la Figura 29 indica la disminución de recursos (región amarilla). En el instante de tiempo 22:35:30 se puede apreciar como el rendimiento es más bajo del esperado (el porcentaje de progreso de la Figura 29 se encuentra en la región que indica que deben aumentarse los recursos). Tras esta

reasignación (visible en el siguiente instante de tiempo de la Figura 31), el trabajo se ejecuta hasta completarse cumpliendo con la calidad de servicio deseada.

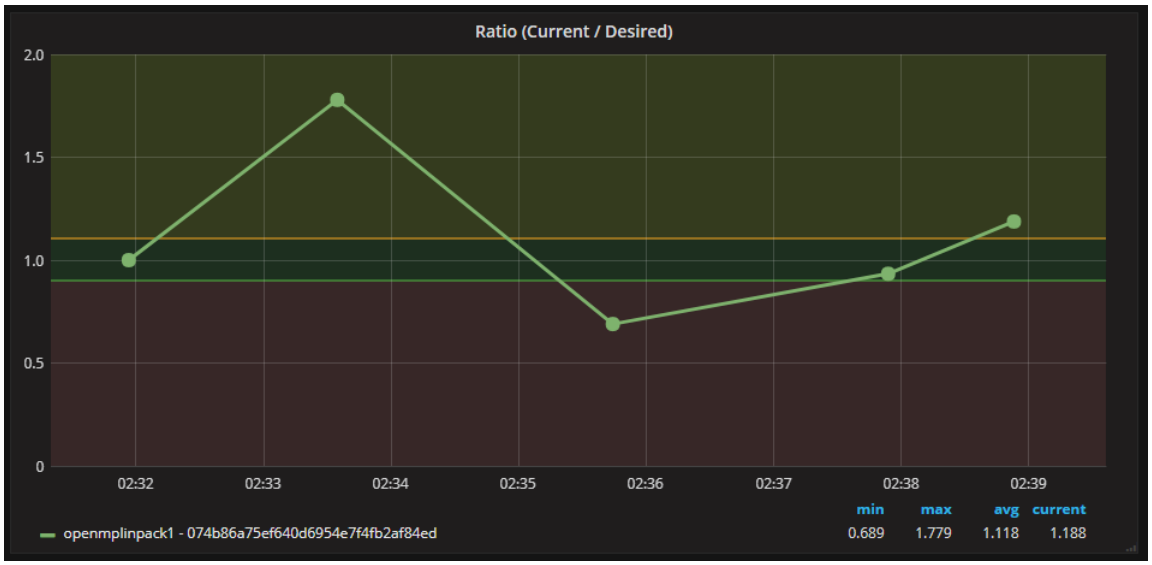

Figura 29: Porcentaje de progreso en función del instante de tiempo en el caso III. Cuando el valor se encuentra en la región amarilla o roja, se produce una disminución o aumento, respectivamente, de los recursos asignados.

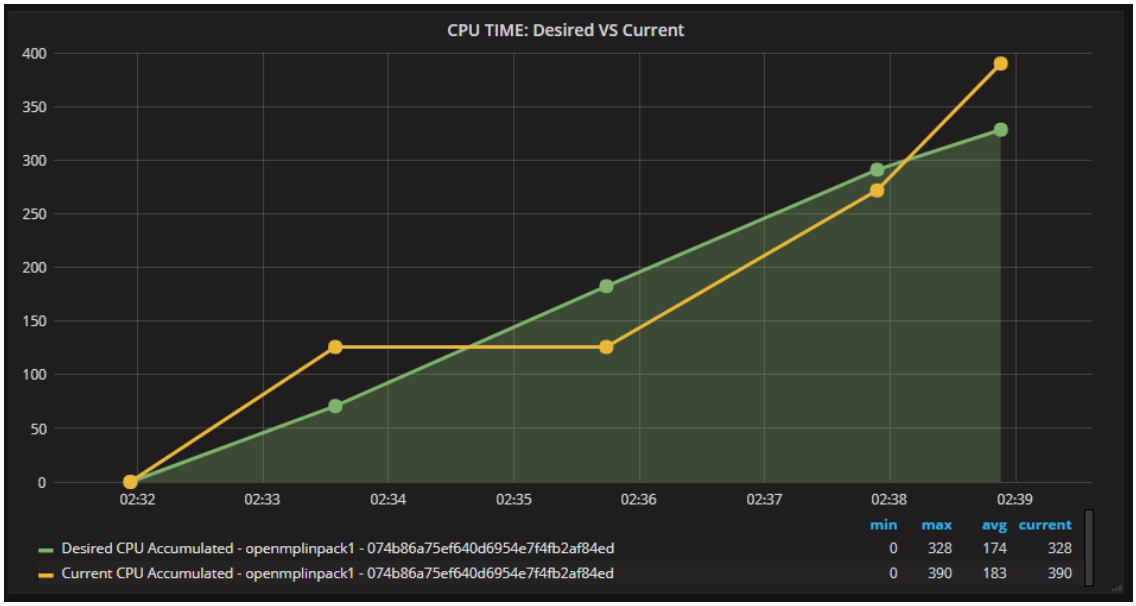

Figura 30: Comparación entre el tiempo consumido real y deseado de CPU en el caso III.

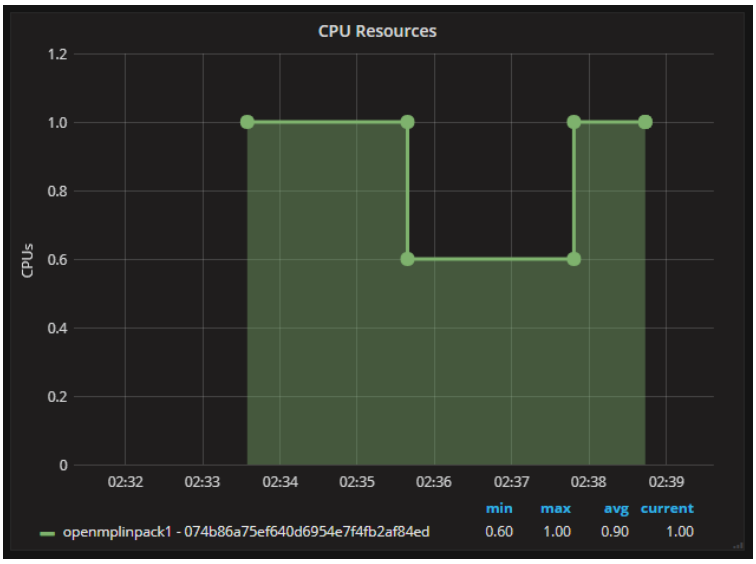

Figura 31: Representa el valor de CPU asignado a la aplicación en el instante de tiempo que acaba de trascurrir para el caso III.

Capítulo 6. Conclusiones

6.1. Conclusiones

El sistema desarrollado en este Trabajo Fin de Máster permite al usuario ejecutar en Mesos trabajos encapsulados en contenedores Docker y, además, que cumplan con una calidad de servicio (QoS). Por lo tanto, se ha alcanzado el objetivo más ambicioso del trabajo.

Para conseguir esto, se han diseñado e implementado dos componentes que tienen como objetivo iniciar la aplicación (*Lanzador*) y realizar la monitorización para ofrecer elasticidad vertical en función del rendimiento (*Supervisor)*.

Gracias a los resultados obtenidos en los casos de estudio, se ha demostrado no solo que es posible, sino que es muy interesante la monitorización de contenedores Docker con el objetivo de variar dinámicamente sus recursos asignados.

Cabe destacar que la disponibilidad de API REST de las herramientas y la utilización de un formato de intercambio de mensajes estándar (JSON) ofrece muchas facilidades a los desarrolladores de aplicaciones. Por este motivo, en futuros proyectos, consideraré todavía más, si cabe, la utilización de formatos y comunicaciones estándar.

Por su parte, las dependencias existentes entre las herramientas utilizadas, han tenido un impacto (no siempre previsto) tanto en la temporalización del desarrollo del trabajo como en la aparición de errores. Concretamente, durante el desarrollo de este trabajo han surgido conflictos debidos a los cambios de versiones de las herramientas usadas (por ejemplo, una nueva versión de Chronos necesitaba un paquete que no estaba soportado en la versión utilizada de sistema operativo).

Finalmente, debe señalarse que la utilización de herramientas que no se encuentran lanzadas de manera oficial al público para aprovecharse de características innovadoras, es un riesgo. En este proyecto, se ha asumido un gran riesgo al utilizar la herramienta nativa de Docker para la creación y restauración de *checkpoints*, ya que solo está disponible en modo *experimental*. Si bien se han conseguido explotar las funcionalidades que presenta, su carácter experimental ha originado irregularidades a las que ha sido necesario adaptarse de forma dinámica en la medida en la que han surgido, no siempre de manera previsible.

En conclusión, se han cumplido todos los objetivos del proyecto, ya que se ha identificado una herramienta de monitorización que pueda integrarse fácilmente en Mesos y se ha implementado un sistema que, dada una especificación de un trabajo encapsulado en un contenedor Docker y una QoS, es capaz de desplegarlo, monitorizarlo y variar los recursos asignados con el objetivo de cumplir con la calidad de servicio acordada.

6.2. Valoración personal

La realización de este trabajo ha sido una gran experiencia muy formativa y motivadora, tanto por lo aprendido en el desarrollo del mismo como por los resultados obtenidos y por haber supuesto un reto en muchos momentos, debido a las consecuencias derivadas de integrar distintas aplicaciones con sus particularidades e incompatibilidades y a los retos que las mismas han originado.

Además, formar parte de un proyecto de investigación que se está desarrollando actualmente, en el que colaboran Europa y Brasil, me ha permitido tanto introducirme en el ámbito de la investigación como obtener una visión global del estado del arte del campo en el que he trabajado. Asimismo, ha sido muy gratificante poder desarrollar mi trabajo en un contexto internacional y mantener contacto con distintos investigadores.

También es importante destacar, como logro personal alcanzado tras la realización de este trabajo, que he ampliado y reforzado los conocimientos en gestión de contenedores Docker, monitorización de sistemas, integración de diferentes tecnologías, API's REST y mejorados lenguajes de programación como Python y bash, así como mi capacidad para la relación interpersonal en el ámbito de la investigación.

6.3. Trabajo futuro

Debido a que este trabajo forma parte del proyecto EUBra-BigSEA, tengo la oportunidad de continuar, de forma inmediata, el desarrollo que conforma este trabajo. Concretamente, durante los meses venideros, participaré en la integración del sistema que he implementado con los desarrollos sobre escalabilidad vertical que forman parte del proyecto. Este sistema contribuirá a proveer de elasticidad vertical a los *frameworks* Chronos y Marathon, lo que forma parte de uno de los objetivos a alcanzar dentro de los previstos en el marco del proyecto EUBra-BigSEA.

Capítulo 7. Bibliografía

[1] Docker- What is a container?
<https://www.docker.com/what-container> [Consulta: 5 de junio de 2017]

[2] Software Projects Built on Mesos.
<http://mesos.apache.org/documentation/latest/frameworks/> [Consulta: 5 de junio de 2017]

[3] HINDMAN BENJAMIN, *et al. (2010). Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*. Berkeley: University of California. <http://mesos.berkeley.edu/mesos_tech_report.pdf> [Consulta: 5 de junio de 2017]

[4] Información sobre las comprobaciones de estado de las tareas en Marathon.
<https://mesosphere.github.io/marathon/docs/health-checks.html> [Consulta: 6 de junio de 2017]

[5] Información sobre el estándar ISO 8601.
<https://www.iso.org/iso-8601-date-and-time-format.html> [Consulta: 6 de junio de 2017]

[6] Documentación de Monasca Openstack.
<https://wiki.openstack.org/wiki/Monasca> [Consulta: 7 de junio de 2017]

[7] Documentación sobre Monasca API.
<https://github.com/openstack/monasca-api/blob/master/docs/monasca-api-spec.md> [Consulta: 7 de junio de 2017]

[8] Fuente de datos soportadas por Grafana Labs.
<https://grafana.com/plugins?type=datasource> [Consulta: 8 de junio de 2017]

[9] Oracle - What Are RESTful Web Services?
<http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html> [Consulta: 8 de junio de 2017]

[10] Introducción a JSON.
<http://www.json.org/json-es.html> [Consulta: 9 de junio de 2017]

[11] Sitio web de Git.
<https://git-scm.com/> [Consulta: 9 de junio de 2017]

[12] The Scrum Guide.
<http://scrumguides.org/scrum-guide.html> [Consulta: 12 de junio de 2017]

[13] Termination Signals.
<https://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html> [Consulta: 19 de junio de 2017]

[14] Docker *plugin* para el Agente Monasca.
<https://github.com/openstack/monasca-agent/blob/master/docs/Plugins.md#docker> [Consulta: 19 de junio de 2017]

[15] DONGARRA, JACK J.*; LUSZCZEK, PIOTR; PETITET, ANTOINE*; *(2003).* "The LINPACK Benchmark: past, present and future" en *Concurrency and Computation: Practice and Experience*, vol. 15, issue 9, p. 803–820).
<http://www.icl.utk.edu/~luszczek/pubs/hplpaper.pdf> [Consulta: 21 de junio de 2017]

[16] Benchmark LINPACK.
<https://people.sc.fsu.edu/~jburkardt/c_src/sgefa_openmp/sgefa_openmp.html> [Consulta: 21 de junio de 2017]
[17] How to collect Docker metrics?
<https://www.datadoghq.com/blog/how-to-collect-docker-metrics/>
[Consulta: 22 de junio de 2017]
[18] Repositorio de este trabajo en GitHub.
<https://github.com/serlophug/Deployment-and-Monitoring-Mesos-Cluster>