Facultad de Ciencias

DESPLIEGUE DE UN ENTORNO DE COMPUTACIÓN *CLOUD* PARA EL EXPERIMENTO CMS

(Deployment of a cloud computing environment for the CMS experiment)

Trabajo de Fin de Máster
para acceder al

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

Autor: Aida Palacio Hoz

Director: Julio Ramón Beivide Palacio

Co-Director: Álvaro López García

Enero - 2018

Agradecimientos

Quisiera aprovechar estas líneas para agradecer a las personas que me han apoyado a lo largo de esta etapa y en el desarrollo de este trabajo.

En primer lugar, me gustaría agradecer a Jesús Marco por darme la posibilidad de realizar este trabajo, de apoyarme y confíar en mí durante estos más de dos años que llevo trabajando con él. Agradecer a mi tutor Mon por querer formar parte de ésto y por todo lo que me ha enseñado en sus clases. También, a mi co-director y compañero Álvaro, ya que acabar este trabajo ha sido, en gran parte por él, gracias por tu supervisión, motivación y sabiduría.

En segundo lugar, gracias al resto de mis compañeros del grupo de Computación avanzada y E-Ciencia por su labor en el Instituto de Física de Cantabria y por ofrecerme ayuda siempre que la necesito. Sobretodo, agradecer, a Miguel Ángel por su disposición y su buen compañerismo.

Quiero agradecer, como siempre, a mis padres, que siempre se preocupan por mí y me apoyan y aconsejan en cada una de mis decisiones.

Gracias a mi amigo, compañero de carrera y trabajo David, por las incontables horas que me aguanta, me aconseja y me ayuda en cada cosa que le pido.

Y también me gustaría agradecer a mi otro David, por su gran apoyo en este poquito tiempo que le conozco, por escucharme, valorarme y por querer estar en todo momento conmigo.

Resumen

El experimento CMS (*Compact Muon Selenoid*) está diseñado para descubrir un rango amplio de partículas y fenómenos que se producen en el Gran Colisionador de Hadrones (LHC). Dentro de este proyecto se realizan análisis de grandes volúmenes de datos sobre diferentes infraestructuras distribuidas de forma global en todo el mundo. Actualmente, el Instituto de Física de Cantabria (IFCA) participa activamente en este proyecto ofreciendo sus recursos de computación. Por un lado, la infraestructura *Grid computing* dónde sus usuarios, así como otros usuarios del proyecto CMS, ejecutan sus simulaciones. Y por otro lado, los recursos locales, como HPC o clústers locales, dónde grupos de investigación nacionales o internacionales, así como usuarios CMS, ejecutan sus análisis.

En los últimos años, los usuarios CMS del IFCA han tendido a ejecutar sus trabajos utilizando otras vías además de la que la infraestructura local les proporciona. Esto suele ser debido a colaboraciones con investigadores que están acostumbrados a trabajar en otros sistemas.

En este trabajo se propone llevar a cabo la implementación y despliegue de un entorno de computación bajo demanda para el experimento CMS sobre una infraestructura *cloud* basada en OpenStack. Este despliegue se llevará a cabo utilizando la solución desarrollada dentro del proyecto europeo INDIGO-DataCloud, llamada DODAS, que consiste en la orquestación de un clúster de computación basado en contenedores y a su vez integrado en la infraestructura de computación global de CMS, *HTCondor*. El objetivo que se persigue es ofrecer un servicio que sea transparente y facilite a los usuarios el uso de la infraestructura local y así, explotar nuestros recursos de manera más eficiente que la utilizada hasta ahora.

DODAS implica la utilización de diferentes componentes y tecnologías que es necesario integrar para lograr el objetivo final del proyecto. En primer lugar, se hace uso del estándar abierto TOSCA como lenguaje para definir la topología necesaria para desplegar la aplicación. Ha sido necesario realizar algunas modificaciones de forma que sea posible utilizar la topología descrita en la infraestructura local. En segundo lugar, se integran diferentes tecnologías y servicios *cloud*. El componente principal es el servicio de orquestación de OpenStack, llamado *Heat*, el cuál es el encargado de manejar todo el ciclo de vida de la infraestructura: desde el despliegue inicial, al escalado de los nodos y finalización del entorno. Como paso final, una vez desplegado el entorno, éste se ha incluído dentro de la *Global Pool* de *HTCondor* para que esta nueva infraestructura pueda ser utilizada por los usuarios de CMS y puedan ejecutar sus simulaciones y análisis sobre ella.

Palabras clave: CMS, *cloud computing*, IFCA, clúster, INDIGO-DataCloud, OpenStack, orquestación, contenedor, TOSCA, *Heat*.

Abstract

The CMS experiment (Compact Muon Solenoid) is designed in order to discover a wide spectrum of particles and phenomena that are generated at Large Hadron Collider (LHC). In this project, large volumen of data is analysed on different distributed infrastructures globally around the world. Nowadays, the Instituto de Física de Cantabria (IFCA) participates actively in this project, offering computing resources. On the one hand, the infrastructure Grid computing, where its users as well as users from CMS project, run their simulations. On the other hand, its local resources, like HPC or local clusters, where national or international researchers, as well as CMS users, run their analysis.

In recent years, the CMS users from IFCA have tended to run their jobs using other ways besides the local infrastructure that is at their disposal.This is due to collaborations with other researchers that work in other systems.

On this project it is proposed to carry out an implementation and deployment of a computing environment on demand for the CMS experiment on a cloud infrastructure based on OpenStack. This deployment will be carried out using the solution developed inside the European project INDIGO-DataCloud, named DODAS, which consists on the orchestration of a computing cluster based on containers, and in turn, integrated in the infrastructure of CMS Global Pool, HTCondor. The main goal is to offer a transparent and more usable service for the users of the local infrastructure in order to exploit our resources in a more efficient way than they used till now.

DODAS implies the use of different components and technologies that are needed to integrate for achieving the final goal of the project. Firstly, the open standard TOSCA is used as language to define the needed topology for the application deployment. It has been necessary to do some modifications in order to be possible to apply the topology described at the local infrastructure. Secondly, it is used different technologies and cloud services that have been needed to integrate as well. The main component is the OpenStack orchestration service, named Heat, which functions are to manage the infrastructure life cycle: from the initial deployment, to node escalation and completion of the environment. Finally, once the environment has been deployed, it has been included inside the CMS Global Pool in order to being used by the CMS users, so that they are able to run simulations and analysis on it.

Keywords: CMS, cloud computing, IFCA, cluster, INDIGO-DataCloud, OpenStack, orchestration, container, TOSCA, Heat.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

El Instituto de Física de Cantabria (IFCA) es un centro mixto, fundado en Junio de 1995, que combina dos instituciones: el Consejo superior de Investigaciones Científicas (CSIC) y la Universidad de Cantabria (UC). El objetivo fundamental del IFCA es la investigación científica en ámbitos como la Astrofísica y la estructura de la materia. Según este objetivo, el Instituto sigue varias líneas de investigación: Física de Partículas, Astronomía y Ciencia del Espacio y Física estadística y no lineal para comprender los componentes de la naturaleza desde las partículas elementales hasta las estructuras del Universo, así como el comportamiento colectivo de la materia.

Dentro de las diferentes líneas de investigación del IFCA, el grupo de computación distribuida y E-Ciencia se centra en la creación de una infraestructura integrada para ser utilizada por investigadores locales e internacionales. Además, contribuye en el campo de *Data Science* a través del análisis de datos en abierto, en diferentes campos de la ciencia y en aplicaciones interdisciplinares aprovechando los recursos computacionales disponibles. Dentro de los diferentes proyectos, internacionales o nacionales en los que el grupo participa, destacan los siguientes según los 2 objetivos principales:

- Análisis de grandes volúmenes de datos usando la infraestructura Tier-2 local dentro del proyecto de **CMS**.
- Integración e implementación de diferentes servicios y aplicaciones en infraestructuras *cloud* en los proyectos **OpenStack**, **EGI-Engage** e **INDIGO-DataCloud**, ambos dentro del proyecto *Horizon2020*, **Lifewatch ESFRI** o **EOSC**.

Hoy en día, la facilidad de acceso a los recursos de computación juega un papel muy importante en el avance científico, ya que permite a los usuarios poder realizar simulaciones que no pueden abordarse ni estudiarse con los métodos tradicionales. Es por esta razón que muchos desarrolladores e incluso, científicos, optan por implementar aplicaciones de uso científico y cada vez hay un mayor número de ellas. Existen numerosos proyectos a nivel nacional e internacional donde colaboran organizaciones e instituciones, como es el caso del grupo de computación distribuida y E-Ciencia del IFCA, con el objetivo de desarrollar soluciones, como pueden ser dichas aplicaciones, que faciliten el trabajo de los usuarios a la hora de realizar sus análisis.

Aun así, el uso de estas soluciones, en muchas ocaciones, no es trivial y es por eso que pueden surgir una serie de problemas. Uno de los principales aparece en el

momento en el que el usuario necesita ejecutar una de estas aplicaciones. No solo eso, que no exista un único usuario sino que sean cien o mil, a los cuales hay que ofrecerles o un sistema donde ellos instalen la aplicación (PaaS o IaaS, términos que veremos en la Sección 2), desarrollada por él mismo o por terceros, o donde sea el administrador el que instale la aplicación y solo les deba dar acceso (SaaS).

La infraestructura local del IFCA, aparte de los nodos de Supercomputación, está compuesta de servidores dedicados a *Grid*, donde se ejecutan diversas aplicaciones, así como servidores donde se integran servicios *cloud* implementados en los proyectos definidos anteriormente. Se puede consultar el Anexo A.1 donde se especifica más detalladamente la infraestructura local. El término *Grid* se refiere a aquellos recursos de cómputo que son mantenidos y configurados por el administrador de la infraestructura y que pueden arrancarse para suplir las necesidades de los usuarios. El principal problema de esto es que muchos de los recursos se mantienen encendidos aunque no se estén utilizando. Por el contrario, el término *cloud computing*, que será explicado más detalladamente en la Sección 2.2 se refiere a un servicio donde el usuario utiliza unos recursos que se arrancan y se apagan dinámicamente en función de las necesidades de las aplicaciones que éste ejecute.

Como se dijo anteriormente, una de las líneas de investigación del IFCA trata de analizar de manera masiva grandes cantidades de datos a través del experimento CMS dentro del LHC. La infraestructura local del IFCA, a través de los servidores *Grid*, les ofrece los recursos de cómputo que necesitan para crear y ejecutar sus simulaciones. Estos servidores son utilizados para ejecutar aplicaciones científicas pero no exclusivamente simulaciones CMS. En base a esto último, hemos visto que estos usuarios utilizan otras infraestructuras, además del *Grid* para ejecutar sus trabajos, como por ejemplo, el servicio de Supercomputación o el servicio *LXPLUS* [2] del Cern. Por lo tanto, sería interesante ofrecer un servicio exclusivo para estos usuarios donde puedan ejecutar sus simulaciones de manera más centralizada. Además, para nosotros, como supervisores y administradores, nos permitiría tener un mayor control del entorno CMS así como de los usuarios que lo utilizan.

1.2. Objetivos

En base a la problemática actual descrita en el apartado anterior, el objetivo principal de este trabajo es crear un servicio *cloud* al que puedan acceder los usuarios CMS del IFCA para ejecutar la fase final de sus análisis. Para la completa instalación del servicio se comenzará configurando un entorno *cloud* que se arrancará y se dejará preparado para instalar y configurar sobre él todos los componentes necesarios para ejecutar aplicaciones CMS. Una vez que todo el entorno esté disponible, sobre él se habilitará una aplicación donde los usuarios accederán para correr sus trabajos. Uno de los requisitos principales, es que todo ello sea transparente para el usuario y se proporcione un entorno flexible manteniendo la eficiencia actual que proporciona nuestra infraestructura.

Por tanto, la idea de este trabajo de fin de máster es desplegar una plataforma sobre una infraestructura *cloud* para uso científico, con el fin de que el usuario ejecute sus análisis de CMS sin necesidad de instalar ni configurar ningún componente para sus simulaciones.

1.3. Estructura de la memoria

La parte principal de esta memoria comienza en el Capítulo 2 donde se explicarán los conceptos y herramientas que se implementan e integran a lo largo de este proyecto. En el apartado 3 se describe la arquitectura de la solución implementada así como todos los pasos a realizar para conseguir integrar la solución final. En este capítulo se explicarán, también, varios problemas encontrados durante la implementación de la solución así como un apartado donde se validará la solución y se visualizarán los resultados obtenidos. Se finalizará la memoria haciendo hincapié en las conclusiones finales a las que se ha llegado y el trabajo futuro a realizar.

Capítulo 2

Antecedentes

En este capítulo se explicará en qué consisten varios conceptos y aplicaciones que se tratarán en la Sección 3.

2.1. Computación científica y experimento CMS

La computación científica se define en [4] como el uso eficiente de los recursos computacionales para desarrollar y resolver problemas numéricos en áreas como la ciencia y la ingeniería. Para resolver estos problemas aplicando este tipo de computación se deben seguir los siguientes pasos:

- Los usuarios crean un modelo matemático de la aplicación que quieren estudiar.
- Desarrollan e implementan algoritmos para resolver los modelos.
- Ejecutan el *software* en un entorno computacional y guardan los resultados.
- Analizan y visualizan los resultados.

La computación científica ha permitido a los usuarios realizar simulaciones y modelos que sustituyen a los métodos tradicionales, como por ejemplo, los análisis exhaustivos en laboratorios. Una de las ventajas de la computación es el ahorro en coste y tiempo que suponen sobre los métodos tradicionales. Además, muchos de los resultados que se pueden obtener a través de la computación son muy variados y muchos de ellos proporcionan avances en distintos campos de investigación, desde la construcción de dispositivos o sensores hasta el estudio de enfermedades.

El experimento CMS (Compact Muon Selenoid) es uno de los dos detectores de partículas de propósito general del Gran Colisionador de Hadrones(LHC), cuyo objetivo es la colisión de haces de protones, situado en el Cern, Suiza. Los experimentos LHC permiten probar la existencia de nuevos procesos o partículas, por lo que los principales objetivos de este detector son el estudio y la búsqueda del *Boson de Higgs* [7], de nuevas partículas que puedan formar nueva materia y de dimensiones espaciales extra.

El experimento CMS está diseñado para descubrir un rango amplio de partículas y fenómenos que se producen en el LHC debido a la colisión de partículas a muy alta energía. Las distintas capas del detector miden datos correspondientes de diferentes partículas. Estos datos son usados, porteriormente, para desarrollar un modelo gráfico

de los eventos que ocurren en la colisión. El *ratio* de eventos que se producen es del orden de 150 Hz, por lo que se deben guardar y procesar varios *PetaBytes*(PB) de datos al año. Al mismo tiempo, el experimento necesita usar recursos computacionales para generar datos simulados que puedan ser analizados en base al objetivo que persiga el usuario dentro del experimento. Para cubrir estas necesidades, las ejecuciones se realizan sobre entornos distribuidos de *Grid* o *cloud computing*.

En el experimento CMS trabajan 4300 físicos, ingenieros, técnicos, estudiantes y personal de soporte de 182 instituciones de 82 paises [1], participando activamente en la actualidad el Instituto de Física de Cantabria (IFCA). El grupo del IFCA, en concreto, ha participado en el desarrollo de un Tier-2 para guardar y analizar todos los datos que se producen durante las colisiones a través de aplicaciones *Grid*. Alrededor de mil científicos de todo el mundo ejecutan sus trabajos en el *Grid* del IFCA, llegando a casi 30.000 *jobs* al día.

2.2. Cloud Computing

Según la definición del *National Institute of Standards and Technology* (NIST) [3], *cloud computing* es un modelo de servicio que permite acceder por red y bajo demanda a un conjunto de recursos de cómputo previamente configurados (red, servidores, almacenamiento, aplicaciones y servicios), que pueden ser rápidamente aprovisionados y lanzados con mínimo esfuerzo de gestión o interacción por parte del proveedor de servicios. Este modelo *cloud* está compuesto de cinco características principales, tres modelos de servicio y 4 modelos de desarrollo que se irán viendo a lo largo de este apartado.

En [4], se define el término *cloud computing* como un nuevo modelo de computación donde el usuario utiliza unos recursos de computación bajo demanda. Así, los clientes del *cloud* pueden acceder a un conjunto de recursos cuando sea necesario, pagando solo su uso sin necesidad de realizar un gasto inicial, como ocurre cuando un usuario u organización utiliza sus propios sistemas. En el Anexo A.1 se explica detalladamente la infraestructura *cloud* del IFCA.

Cloud computing tiene las siguientes cinco características principales:

- **Servicio bajo demanda**: Un usuario puede hacer una petición de un conjunto de recursos de cómputo, como tiempo de cpu y almacenamiento, según lo requiera, sin tener que interaccionar con el proveedor del servicio.
- **Acceso por red**: Los recursos de cómputo están disponibles por red y los usuarios acceden a ellos a través de diferentes mecanismos usando plataformas de todo tipo, desde móviles o *tablets* hasta portátiles o *workstations*.
- **Agrupación de recursos**: Los recursos de cómputo son agrupados siguiendo el modelo *multi-tenant*, de manera virtual o física y asignados dinámicamente según la demanda del usuario. Este modelo proporciona una capa de abstracción para el usuario creando una sensación de independencia aunque el usuario no tenga conciencia de dónde están localizados los recursos exactamente.
- **Elasticidad**: La capacidad de los recursos es escalable, de manera que se puedan proveer bajo demanda del usuario. Así, la capacidad disponible aparece como ilimitada y puede ser incrementada en cualquier momento.

[1] Datos de Febrero de 2014

- **Monitorización**: Los sistemas *cloud* controlan y optimizan recursos en base a la carga y el uso de los mismos (almacenamiento, ancho de banda, usuarios activos, etc). El uso de los recursos es monitorizado, controlado y posteriormente, reportado de manera transparente al usuario y al proveedor del servicio utilizado.

2.2.1. Modelos de Cloud Computing

A la hora de ofrecer servicios de *cloud computing*, se pueden diferenciar tres modelos de servicio como se ve en la Figura 2.1: *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) y *Software as a Service* (SaaS). En el caso de los modelos de servicio, el usuario y el proveedor del servicio *cloud* deciden cuál es la solución que se requiere para desplegar y gestionar sus aplicaciones.

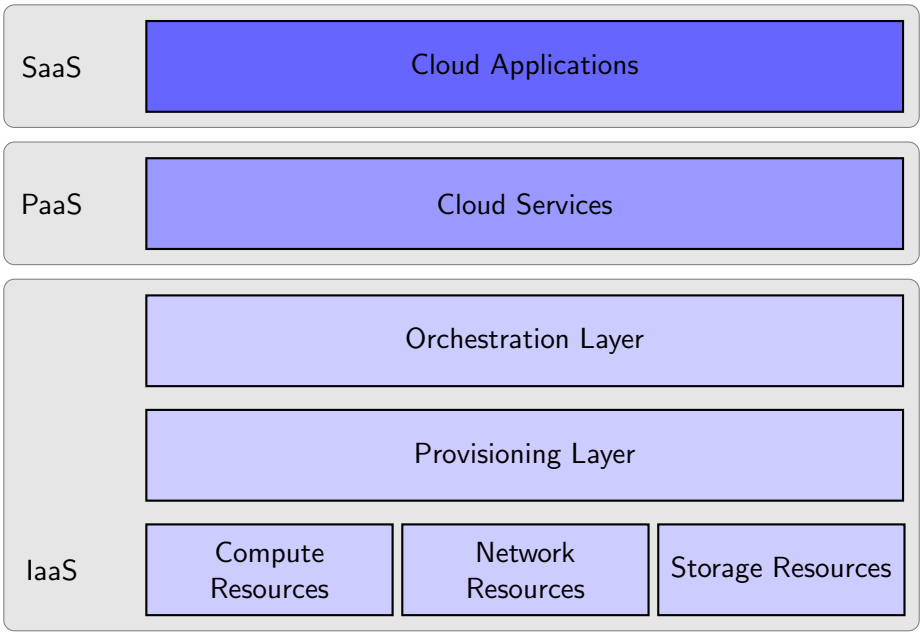

Figura 2.1: Arquitectura servicios *cloud computing*. Fuente [4]

.

- ***Software as a Service* (SaaS)**: El cliente o usuario tiene acceso a una aplicación software con todas las funcionalidades que éste requiera. Esta aplicación está instalada sobre una infraestructura y puesta a disposición del usuario a través de internet.

- ***Platform as a Service* (PaaS)**: El cliente desarrolla sus propias aplicaciones sobre un sistema operativo que cuenta con todas las herramientas de desarrollo para que el usuario pueda realizar sus soluciones. La plataforma está alojada sobre una infraestructura, a la cual el usuario no tiene ningún tipo de control.

- ***Infraestructure as a Service* (IaaS)**: El cliente tiene acceso al almacenamiento, la red y otros recursos de cómputo. Sobre este servicio, el usuario alojará sus aplicaciones y plataformas y solo tendrá completo control sobre esta última, no sobre la infraestructura.

Una infraestructura *cloud* (IaaS) es una colección de hardware y software combinando la capa física con la capa abstracta donde únicamente el proveedor de servicio o administrador IT tiene control sobre él y decide a quién debe dar acceso. Es sobre esta capa donde se desarrollan todos los servicios *cloud* (PaaS), ya sean servicios de red o de autenticación y las aplicaciones (SaaS), que a su vez se alojan sobre estos servicios.

En este trabajo nos vamos a centrar en estas últimas capas: PaaS y SaaS. En el Capítulo 3 se explicará con un mayor detalle cómo se ha realizado el despliegue de un entorno en máquinas virtuales sobre una infraestructura utilizando un servicio de orquestación (PaaS).

De cara a las diferentes formas de desplegar toda la infraestructura *cloud*, existen cuatro modelos de desarrollo:

- ***Private Cloud:*** La infraestructura *cloud* se provee a una única organización con múltiples clientes.
- ***Community Cloud***: La infraestructura se provee para el uso exclusivo de una comunidad de clientes desde organizaciones que tienen mismos intereses o un conjunto de organizaciones.
- ***Public Cloud:*** La infraestructura se provee para uso público. Puede ser para uso propio o para uso en un negocio, uso académico o una organización de gobierno.
- ***Hybrid Cloud:*** Combinación de una o más infraestructuras que están estandarizadas conjuntamente por una tecnología.

Por parte de los operadores de IT, la adopción de tecnologías *cloud* ofrece grandes beneficios como la disponibilidad de servicios *cloud* y la utilización y pago de recursos según la demanda del usuario. Hoy en día, está incrementando el número de aplicaciones científicas con un propósito cada vez más especializado. Muchas de ellas son usadas por un conjunto de usuarios que no pertenecen a una misma comunidad y es necesario que exista **interoperabilidad** entre las distintas infraestructuras. El término interoperabilidad radica en poder intercambiar información y utilizar dicha información entre dos sistemas u organizaciones dispares entre sí, con el único fin de obtener un beneficio. En este caso, poder dar acceso a un servicio a los usuarios de ambas comunidades.

2.2.2. OpenStack

OpenStack [9] es un software *open source* para la creación de *clouds* públicos y privados. OpenStack se encarga del control de un conjunto de recursos de cómputo, almacenamiento y red de un *DataCenter* (IaaS). El acceso a los recursos se realiza a través de un *dashboard*, vía interfaz web, que permite a los administradores controlar los recursos y dar acceso a los usuarios a gran parte de los mismos para que puedan crear sus propias aplicaciones. En la Figura 2.2 se muestran, de manera genérica los diferentes componentes a los que da acceso OpenStack, tanto recursos de cómputo (*Virtual Machines* o *Containers*), de almacenamiento (*Object Storage* o *Block Storage*), de red y las herramientas de monitorización y control de los recursos.

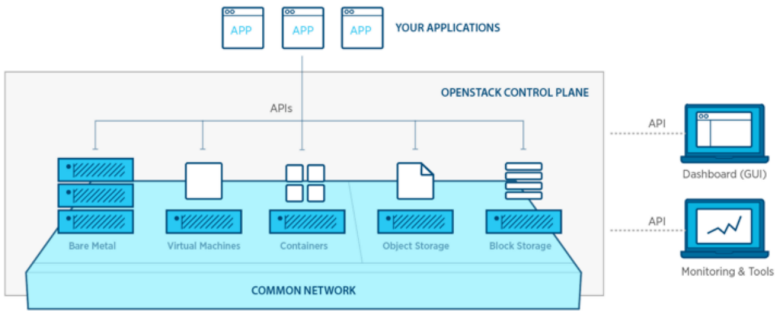

Figura 2.2: Arquitectura de Openstack. Fuente: *Openstack*

En el proyecto OpenStack participan 82.775 miembros de 187 países del mundo dónde contribuyen para construir una plataforma de *cloud computing* con más de 20 millones de líneas de código.

Para la administración de todos los recursos *cloud*, según la versión de OpenStack se ponen a disposición una serie de servicios que se explicarán con más detalle en el siguiente apartado.

Servicios de Openstack

Para la gestión y el control de todos los recursos que se muestran en la Figura 2.2, el administrador de la infraestructura de Openstack puede instalar una serie de servicios. La API de OpenStack se agrupa en varias secciones dependiendo de los recursos que se quieran configurar. Según la última versión, *Ocata*, aquí resumo los más importantes:

- CÓMPUTO:
  - Nova : Provee control y acceso, bajo demanda, a recursos de computo para la creación de máquinas virtuales o contenedores.
  - Glance : Permite registrar y administrar imágenes para la creación de máquinas virtuales. Estas imágenes pueden ser guardadas en almacenamiento con distinto tipo de Sistema de Ficheros como el que provee el servicio *Cinder*, basado en un sistema de ficheros de almacenamiento en bloques.
- ALMACENAMIENTO
  - Swift : Provee almacenamiento en grandes bloques de datos que no cuentan con ningún tipo de estructura. Según OpenStack, el almacenamiento de objetos se realiza de manera eficiente, distribuida y con una gran disponibilidad de acceso.
  - Cinder : Servicio de almacenamiento en bloques. Provee una API para que el usuario que acceda a los recursos de almacenamiento no necesite

conocer cómo están distribuidos los datos ni en qué tipo de dispositivo de almacenamiento se encuentran.

- RED
  - Neutron  : Servicio que presta un servicio de red SDN para que las máquinas virtuales tengan un acceso a la red.
- AUTENTICACIÓN
  - Keystone  : Provee una API para que los usuarios puedan autenticarse en OpenStack y acceder a los recursos *cloud* del sistema. Soporta LDAP, OAuth, OpenID Connect, etc.
- HERRAMIENTAS DE GESTIÓN
  - Horizon  : Interfaz web que ofrece un *dashboard* desde el que los usuarios pueden acceder y pedir los recursos *cloud* que necesiten, creando instancias de máquinas virtuales, administrando la red, orquestando máquinas, etc.
  - Openstack Client (CLI): Ofrece una API para administrar los recursos a través de la línea de comandos.
- APLICACIONES
  - Heat  : Servicio que osquesta un conjunto de recursos para la instalación de una aplicación *cloud*. Se explicará con más detalle en el siguiente apartado.
  - Murano  : Permite a los administradores de una infraestructura crear un catálogo con el conjunto de aplicaciones disponibles.

2.3. Orquestación de recursos

Según la Real Academia Española (RAE), **orquestar** es instrumentar una composición para una orquesta. Así mismo, instrumentar es preparar unas partituras de una composición musical para cada uno de los instrumentos que la ejecutan. Si llevamos este término a nuestro caso de uso, en la Figura 2.3 veremos que el objetivo es el mismo: instalar, configurar y preparar un conjunto de componentes de cómputo, almacenamiento y red, para albergar una aplicación sobre ellos.

Existen varias herramientas y servicios de orquestación. La limitación de muchos de ellos es que se han desarrollado para una plataforma *cloud* y solo son soportados para dichas plataformas. Este proyecto se realizará utilizando la herramienta de orquestación proporcionada por OpenStack, **Heat**, y su correspondiente lenguaje de orquestación llamado *Heat Orchestration Template* (HOT) basado en el lenguaje de código abierto YAML [10]. El proyecto OpenNebula [34] provee una herramienta de orquestación basada en JSON llamada *Oneflow*. El servicio *Eucalyptus* de Amazon soporta orquestación gracias a un servicio web llamado *AWS CloudFormation* [13].

Figura 2.3: Arquitectura de la orquestación.

Apache CloudStack [14] es un *software* que actúa como una plataforma de orquestación para desplegar una Infraestructura (IaaS) pública o privada [15]. CloudStack puede usar diferentes hipervisores como KVM, vSphere, XenServer, VMWare y OracleVM para virtualizar el sistema y soporta la API de *Amazon Web Services* (AWS) además de la suya propia.

En el proyecto OpenStack es **Heat** el servicio encargado de la orquestración de un conjunto de recursos de una infraestructura para la creación automática de una aplicación *cloud*. Esta orquestación se basa en un *template* donde se define la topología de la aplicación. Heat, además, provee una API nativa de OpenStack y una API compatible con *AWS Cloudformation*.

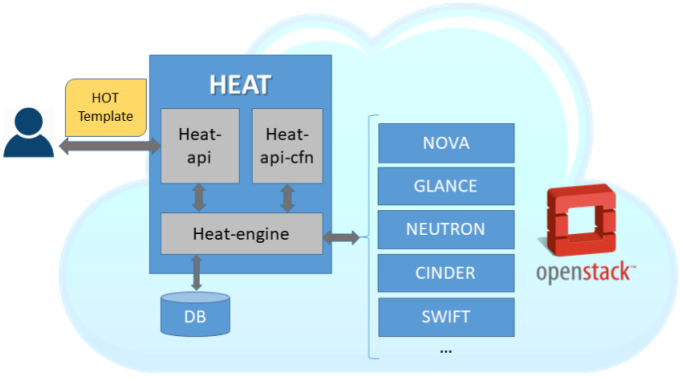

Figura 2.4: Arquitectura de HEAT.

Como se muestra en la Figura 2.4, el usuario crea un *template* donde define toda la topología de la aplicación que va a desplegar y, a través del cliente de OpenStack, se lo manda a Heat para que realice la orquestación de la infraestructura. Heat tiene 3 componentes principales: *heat-api*, *heat-api-cfn* y *heat-engine*. ***Heat-api*** es una API de OpenStack encargada de procesar las peticiones del usuario y enviárselas a *heat-engine*. ***Heat-api-cfn*** es una API compatible con *AWS CloudFormation* que procesa las peticiones y se las envía a *heat-engine*. ***Heat-engine*** es el componente más importante

y es el encargado tanto de operar con el resto de servicios de OpenStack para desplegar la aplicación, como de responder eventos de los otros dos servicios.

2.3.1. HOT

Para crear la topología de la aplicación, el *template* tiene que estar escrito en un lenguaje entendible por Heat. Es *Heat Orchestration Template*(HOT) el lenguaje soportado por Heat para orquestar toda la infraestructura.

A continuación, en *Listing* 2.1, se describe un ejemplo en HOT donde se definen todas las partes de las que debe componerse el *template* para su posterior orquestación.

Listing 2.1: HOT example.

```
heat_template_version: 2013-05-23

description: HOT Template simple example

resources:
  server:
    type: OS::Nova::Server
      properties:
        flavor: c4.large
        image: linux-ubuntu-14.04
```

El primer parámetro, `heat_template_version` define la versión de Heat. Esta versión debe ser válida según la lista con todas las versiones de HOT [11]. *Description* es opcional pero es interesante para que el usuario pueda entender en qué consiste la aplicación.

La sección que comienza por `resources` es obligatoria y debe contener al menos un recurso. En este caso, se trata del despliegue de un recurso de cómputo, un servidor Ubuntu con una capacidad según el sabor `c4.large`. Este sabor viene definido en el servicio de `Nova` de OpenStack y dependiendo de su valor pueden variar el número de cpus y la cantidad de RAM y disco.

Se pueden añadir también parámetros al fichero que funcionen como variables de entrada para los recursos. La ventaja de tener estos parámetros ya definidos al principio permite que varios recursos puedan reusarlos. Dicha sección comienza por la variable `parameters`. Por ejemplo, los valores `flavor` e `image` del ejemplo anterior 2.1 pueden definirse en la sección `parameters` y el recurso `server` accederá a estos valores a través de la llamada `get_param`. Esto se muestra en el *Listing* 2.2.

Según el tipo de recurso, Heat necesita conectarse con otros servicios de OpenStack para poder crear la infraestructura. En la infraestructura local del IFCA se alojan los servicios especificados en el Anexo A.2.

Conexión de Heat con otros servicios de OpenStack en HOT

Para crear una nueva máquina virtual necesitamos que el servicio *Nova* cree un nuevo servidor con las capacidades que se hayan definido en el *template*. El parámetro `type` indica qué tipo de recurso se va a tratar y con qué servicio, Heat, va a necesitar conectarse. En el ejemplo, Heat necesita conectarse con el servicio de Nova para crear un nuevo `server`.

Listing 2.2: HOT example Parameters.

```
parameters:
    image_server:
      type: string
      default: linux-ubuntu-14.04
    flavor_server:
      type: string
      default: cm4.xlarge
resources:
   server:
    type: OS::Nova::Server
    properties:
      name: mesos-master-server
      flavor: { get_param: flavor_server }
      image: { get_param: image_server }
```

En el ejemplo que se muestra en el *Listing* 2.3, se trata de crear una nueva regla en el *firewall* (llamado *security group*) para abrir un puerto y poder acceder al servidor vía ssh.

Como se muestra en el siguiente ejemplo, se puede crear un nuevo volumen para almacenar los datos del servidor. Para ello, *heat-engine* necesita conectarse con el cliente de Cinder como se ve en *Listing* 2.4 con la variable `type` dentro del recurso `server_volume`.

Además, de los tipos de recursos que se han explicado en los ejemplos, existen otros documentados en [12] para la versión 9.0.2.dev3. Por ejemplo, para crear un nuevo usuario en el sistema, es necesario que *heat-engine* se conecte con el servicio de autenticación Keystone. Por tanto, el tipo de recurso que hemos de definir es `OS::Keystone::User`. O por ejemplo, para asignar una nueva dirección IP pública o flotante al servidor para que el usuario pueda acceder a la máquina, *heat-engine* accede al servicio de Nova y por tanto el nuevo recurso es del tipo `OS::Nova::FloatingIP`. El usuario o la dirección flotante se agregaría al recurso `server` a través de la función `get_resource`:

```
floating_ip: { get_resource: resource_floating_ip }
```

Al igual que se pueden añadir parámetros de entrada, se pueden añadir parámetros de salida. La funcionalidad de dichas variables es que los usuarios puedan tener acceso a ellos una vez se termine el despliegue de la infraestructura. Para crear estas variables es necesario crear una nueva sección que comience con `outputs`. En el ejemplo que se muestra en *Listing* 2.5, si se quiere instalar una aplicación a la que el usuario necesita tener acceso, sería necesario devolver la URL donde esté escuchando dicha aplicación a través de la variable `template`.

2.3.2. TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) es un estándar abierto desarrollado por OASIS [8] que se usa como lenguaje para definir la

Listing 2.3: HOT example Neutron.

```
heat_template_version: 2013-05-23

description: HOT Template simple example Neutron

resources:
   server:
    type: OS::Nova::Server
    properties:
      name: mesos-master-server
      flavor: cm4.xlarge
      image: { get_param: imageCMS }
      networks:
        - port: { get_resource: server1_port }

  server1_port:
    type: OS::Neutron::Port
    properties:
      network_id: { get_param: networkCMS }
      security_groups: [{ get_resource:
                            server_security_group }]

  server_security_group:
    type: OS::Neutron::SecurityGroup
    properties:
      description: Test group to demonstrate
                   Neutron security group
                   functionality with Heat.
      name: test-security-group
      rules: [
        {remote_ip_prefix: 0.0.0.0/0,
        protocol: tcp,
        port_range_min: 22,
        port_range_max: 22},
        {remote_ip_prefix: 0.0.0.0/0,
        protocol: icmp}]
```

Listing 2.4: HOT example Cinder.

```
heat_template_version: 2013-05-23
...
resources:
  ...
  server_volume:
    type: OS::Cinder::Volume
    properties:
      availability_zone: String
      image: String
      volume_type: String
```

Listing 2.5: HOT example outputs.

```
heat_template_version: 2013-05-23
...
outputs:
  app_url:
    description: URL of the deployed application
    value:
      str_replace:
        template: http://app:8080
```

topología de una aplicación y la infraestructura de un conjunto de servicios *cloud*, las relaciones entre las diferentes partes del servicio y las operaciones que realizan dichas aplicaciones (despliegue, apagado, etc). TOSCA es soportado por múltiples compañías, ya sea como contribuidores, usuarios o desarrolladores. Entre estas compañias destacan: AT&T, Bank of America, Brocade, Cisco, Fujitsu, GigaSpaces, Huawei, IBM, Intel, Red Hat, SAP, VMWare, Vnomic y ZTE. Los propósitos principales de TOSCA son:

- Portabilidad de los despliegues en cualquier servicio *cloud*.
- Migración de servicios existentes a las infraestructuras *cloud*.
- Flexibilidad a la hora de desplegar las aplicaciones.

A la hora de describir la topología del entorno que se quiere en TOSCA se debe crear un nuevo fichero que conste de los dos bloques principales: los nodos y las relaciones entre ellos. Un nodo representa cualquier componente de la infraestructura como un sevidor, la red o un servicio. Mientras tanto, las relaciones describen cómo se conectan los nodos entre sí. Este *template* debe estar correctamente definido para ser enviado y entendido por el orquestador que se quiera utilizar para el despliegue.

Cada nuevo nodo creado viene definido por un tipo, los más usados son los siguientes:

- **Compute:** representa un servidor donde se alojan y ejecutan los servicios o las aplicaciones.

- **WebServer**: componente *software* abstracto que puede alojar una o más aplicaciones web.
- **WebApplication:** representa una aplicación web que corre en un *Web Server*.

Los ficheros de TOSCA son escritos en formato YAML [10] ya que su sintaxis es mucho más entendible para el usuario que, por ejemplo, XML [35] y son conocidos como *TOSCA Simple Profile in YAML*. La versión actual es 1.1.

A continuación, se explicarán las partes de las que debe constar un fichero TOSCA y en qué consiste cada una de ellas con un ejemplo sencillo.

Listing 2.6: TOSCA example.

```
tosca_definitions_version: tosca_simple_yaml_1_0

description: TOSCA Template simple example

topology_template:

  node_templates:
    server:
      type:tosca.nodes.Compute
      capabilities:
        host:
          properties:
            num_cpus: 2
            mem_size: 2 GB
            disk_size: 50 GB
        os:
          properties:
            image: linux-ubuntu-14.04
```

El *Listing* 2.6 describe un entorno con un único servidor Ubuntu 14.04. El fichero TOSCA comienza definiendo `tosca_definitions_version`, `description` y `topology_template`. El primer parámetro indica la versión de TOSCA YAML en la que se basará el desarrollador para definir en TOSCA todo el entorno. El siguiente parámetro, como su propio nombre indica, describe brevemente en qué consiste la infraestructura a desplegar. La parte más importante comienza con la variable `topology_template` donde se define toda la topología de la infraestructura.

TOSCA Simple Profile tiene una serie de tipos de nodo por defecto para crear diferentes tipos de servicios. En el caso del ejemplo, se define un nodo `Compute` para crear un entorno con un único servidor además de dos variables `host` y `os` dentro de las capacidades del nodo. La ventaja de TOSCA es que el orquestador entiende todos estos conceptos y es capaz de desplegar la infraestructura sin que el desarrollador tenga que implementar nuevo código para crear cada uno de los componentes. El orquestador crea la infraestructura del servidor con los requisitos que se hayan definido en las propiedades de cada una de las capacidades. Así, en `host` se describen aquellas opciones que el nodo necesita para poder arrancar sus aplicaciones como el número de cpus, la cantidad de memoria y disco, etc. En `os` se crean las propiedades referentes al sistema operativo que el servidor necesita tener para poder ser instanciado.

El siguiente diagrama de la Figura 2.5 muestra en qué consiste el ejemplo anterior: Un servidor Ubuntu 14.04 con 2 CPUs, una memoria de 2GB y un tamaño de disco de 50 GB.

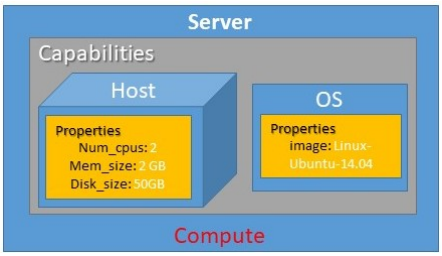

Figura 2.5: TOSCA *Example Template*: *Compute Node*

Gracias a la definición de toda la infraestructura en el fichero de TOSCA se podrá realizar todo el proceso de despliegue de manera automática sin tener que ir realizando paso a paso la instalación, configuración, control, etc.

TOSCA Parser

TOSCA Parser es, también, un proyecto de OpenStack aunque no está restringido únicamente a usarse dentro del mismo ya que es una herramienta de propósito general con licencia de *Apache 2*. TOSCA Parser es una librería de Python que trata de parsear *TOSCA Simple Profile in YAML* explicado en la Sección 2.3.2 anterior, *TOSCA Cloud Service Archive* (CSAR) y *TOSCA Simple Profile for Network Functions Virtualization*(NFV) creando una representación en memoria de los nodos y sus relaciones como se ilustra en la Figura 2.6 .

Heat Translator

Heat Translator [6] es una herramienta desarrollada en el proyecto OpenStack que traduce ficheros escritos en un lenguaje diferente a HOT para obtener un *template* en HOT. En la actualidad, Heat Translator tiene el objetivo de traducir un *template* de TOSCA a HOT, por lo que en este trabajo, utilizaremos esta herramienta para poder desplegar la infraestructura en el *cloud* de OpenStack. Aun así, esta herramienta se podría usar para traducir cualquier formato que no sea TOSCA, implementando los *plugins* correspondientes.

Heat Translator puede ser ejecutado tanto desde línea de comandos como desde la interfaz web de OpenStack. Además, está bien integrado en distintos proyectos de OpenStack y por eso, es usado en varios de ellos, como *OpenStack NFV Orchestration project*, Tacker [16].

Heat Translator toma como entrada la representación creada por TOSCA Parser y lo mapea a recursos de Heat para producir el *template* de HOT.

2.4. Docker

Docker [17] es una plataforma que tiene como principal objetivo la ejecución de aplicaciones en forma de contenedores. Un contenedor es un paquete software que contiene todo lo necesario para poder ejecutar la aplicación: librerías del sistema, código,

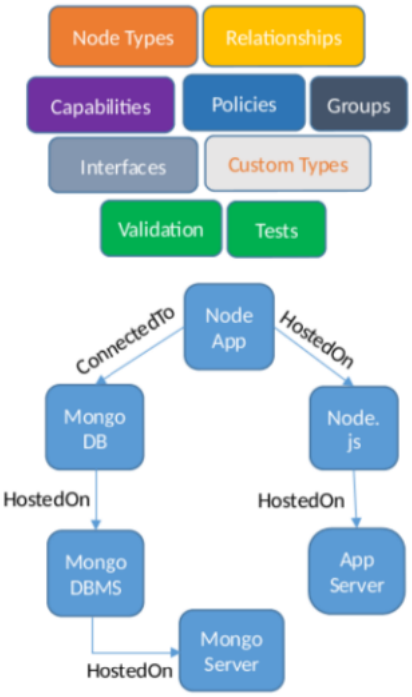

Figura 2.6: Representación de nodos en TOSCA Parser

configuración y herramientas del sistema. También se podrían definir los contenedores como máquinas virtuales "ligeras" ya que comparten el núcleo del sistema operativo con la máquina donde se está ejecutando, la diferencia es que se ejecutan de manera aislada sobre un sistema de ficheros independiente. Ésto hace posible que no sea necesario virtualizar el Sistema Operativo (SO) base provocando que el consumo de recursos sea menor y el despliegue de las aplicaciones más rápido.

Las dos ventajas de tener nuestras aplicaciones en contenedores es poder ejecutarlas independientemente de la plataforma y poder reducir conflictos entre usuarios que estén ejecutando distintas aplicaciones en la misma plataforma.

¿Por qué usar docker y no Máquinas Virtuales?

Como se ha explicado anteriormente, un contenedor es una abstracción de la capa de la aplicación que empaqueta tanto el código como las dependencias. Cada contenedor se ejecuta en la misma máquina sobre el kernel del sistema operativo como si fuera un proceso independiente. Una máquina virtual, en cambio, es una abstracción de la capa *hardware* que hace posible ejecutar varios servidores sobre un único servidor físico a través de un *hypervisor*. Como se observa en la Figura 2.7, debido a que una VM incluye una copia del SO, un conjunto de aplicaciones con sus correspondientes librerías, ésta genera una imagen de mayor tamaño y por tanto supone que el arranque sea mucho más lento que en el caso del contenedor, que es casi instantáneo.

Por tanto, Docker facilita al administrador o al usuario del sistema el despliegue y la ejecución de las aplicaciones sin tener que instalar todas las dependencias. Esta es una de las razones por las que en la solución propuesta se ha decidido encapsular cada una de las aplicaciones en contenedores Docker.

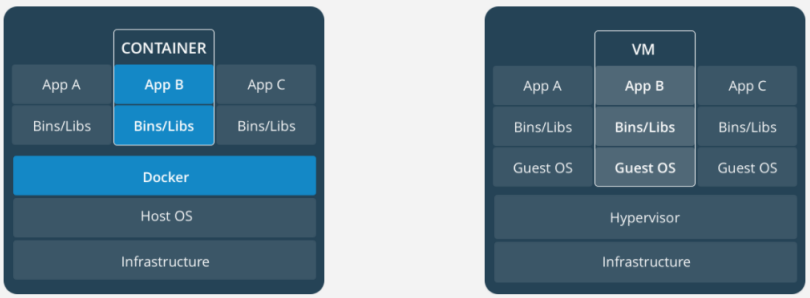

Figura 2.7: Arquitectura Docker (izquierda) vs Máquina Virtual (derecha)

2.5. INDIGO-DataCloud

INDIGO-DataCloud (INtegrating Distributed data Infrastructures for Global ExplOtation) [21] es un proyecto europeo fundado bajo el programa *Horizon2020* y liderado por el Istituto Nazionale di Fisica Nucleare(INFN). INDIGO-DataCloud está compuesto por instituciones como el CSIC, la Universidad Politécnica de Valencia (UPV), el *Istituto Nazionale di AstroFisica*(INAF) o el Cern y empresas como *T-Systems* o INDRA. Todas estas instituciones y empresas han trabajado juntas para desarrollar y proveer las soluciones propuestas dentro del proyecto y cubrir las necesidades de la comunidad científica a nivel Europeo.

El objetivo principal de este proyecto es el desarrollo de una plataforma de datos y de computación para la comunidad científica que se despliega sobre múltiples e-Infraestructuras *Clouds* (IaaS) híbridas. Estas infraestructuras van desde *clouds* privados, como pueden ser OpenStack Newton y OpenNebula [34] hasta *clouds* públicos, como AWS o T-Systems Open Telekom Cloud. Hasta ahora, INDIGO-DataCloud ha desarrollado, validado y publicado un gran número de componentes cubriendo los tres modelos de servicio *cloud* (IaaS, PaaS y SaaS). Estos componentes se han aplicado e implementado en 15 casos de uso reales cubriendo cuatro áreas diferentes: *Environmental and Earth science*, *Biological and Medical science*, *Social science and Humanities* y *Physics and Astrophysics*. Todo el *software* desarrollado en INDIGO es código abierto además de contribuir en el desarrollo y en la implementación de estándares.

Muchos de los componentes desarrollados en INDIGO han sido adoptados en proyectos como OpenStack, OpenNebula, OneData y TOSCA *adaptor*. INDIGO también colabora con grandes compañías como ATOS, INDRA, T-Systems, IBM y otras para facilitar la adopción de sus componentes. Las tecnologías utilizadas dentro del proyecto son actuales y populares dentro de la comunidad científica y comercial por lo que cualquier miembro de estas comunidades puede ayudar a mejorar sus soluciones.

2.6. Conclusiones

El proceso que debe seguir el usuario para instalar una aplicación, en su propio servidor, consiste en 6 pasos principales, como se muestra en la Figura 2.8:

- **Provisión**: Reservar una serie de recursos (cpu, memoria, disco...) según los requisitos de la aplicación.

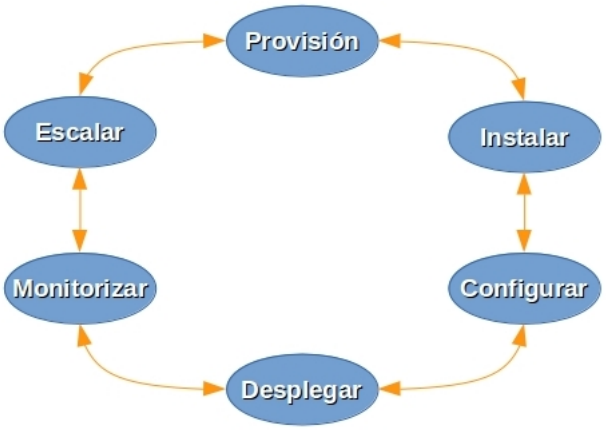

Figura 2.8: Proceso de instalación de una aplicación.

- **Instalación**: Establecer una aplicación en un servidor o infraestructura. En este paso se incluiría el lugar donde descargaríamos la aplicación y, posteriormente, la descarga de la misma.
- **Configuración**: Adaptar las opciones de la aplicación para conseguir ejecutarla como requiere el usuario.
- **Despliegue**: Dejar lista la aplicación para poder ser usada.
- **Monitorización**: Controlar que todo funciona como se requiere.
- **Escalabilidad**: Adaptar y preparar el servicio para ser utilizado por más usuarios. En este término hay que tener en cuenta que si queremos dar el mismo servicio a todos los usuarios, la calidad del servicio no puede disminuir a medida que accedan más clientes. Por tanto, puede tener que repetirse el primer paso y realizar una provisión mayor de recursos.

Para abordar este problema se han planteado una serie de cuestiones: *¿Cuánto tiempo tarda cada usuario en instalar y configurar el software para adaptar y, posteriormente, ejecutar su simulación? ¿Es realmente necesario que el usuario deba hacer todo este proceso? ¿Cuál es la parte donde el usuario pierde más tiempo a la hora de ejecutar sus simulaciones? ¿Y si, únicamente, tiene que lanzar su simulación sin preocuparse de cómo se instaló todo el software?*

Una de las tareas más costosas para los usuarios es tener que crear una configuración apropiada para el entorno donde necesitan hacer sus simulaciones. Dependiendo los resultados que quieran obtener así como las simulaciones que necesiten correr es necesario adaptar el entorno. Por tanto, la solución es tener una configuración común donde los cambios para cada usuario sean los mínimos. Así, se conseguirá automatizar todos los pasos que realiza el usuario y que se muestran en la Figura 2.8 en uno único.

En vista de la solución que se quiere ofrecer para solventar el problema planteado arriba y en el capítulo anterior, en este capitulo se ha tratado de contextualizar de manera general la solución que se explicará en el capítulo siguiente. Primero, he explicado el proyecto CMS para el que se plantea el problema. En la Sección 2.2 se

ha introducido el término *cloud computing* así como los diferentes modelos de servicio que se pueden implantar en un *DataCenter* como el que tenemos instalado en el IFCA. También se define el proyecto OpenStack en el que se basan todos los servicios que se han usado para implantar nuestra solución. En el siguiente apartado, se ha comenzado definiendo qué es la orquestación de recursos así como en qué consisten los lenguajes HOT, utilizado por el servicio Heat de Openstack, y TOSCA. Como se explicará en el siguiente capítulo, primero, es necesario describir todos los componentes de los que constará el nuevo servicio, y para ello se utiliza tanto TOSCA como HOT. Una vez tengamos definidos los componentes de la solución, se usarán contenedores Docker para el despliegue de cada uno de ellos. Por último, se ha descrito en qué consiste el proyecto INDIGO-DataCloud donde se engloba la solución que se va a integrar.

Capítulo 3

Implementación del servicio cloud de CMS en OpenStack Heat

En este apartado se va a explicar cómo se ha llevado a cabo el despliegue de un entorno de computación para el experimento CMS sobre una infraestructura *cloud* basada en OpenStack. Para la realización de este despliegue se ha utilizado la solución desarrollada para el proyecto INDIGO-DataCloud que consiste en la orquestación de un clúster de Mesos para ser integrado en la infraestructura global de computación de CMS [5].

3.1. Solución de INDIGO-DataCloud: DODAS

Dentro de las diferentes líneas que se desarrollan en el proyecto INDIGO-DataCloud se ha creado una nueva herramienta llamada DODAS (*Dynamic on Demand Analysis Service*) implementada sobre los servicios que ofrece INDIGO-DataCloud. El objetivo del servicio es generar un clúster de manera automática y bajo demanda, concretamente un clúster basado en contenedores *Docker*, concepto explicado anteriormente en la Sección 2.4. DODAS se apoya sobre otras soluciones de INDIGO-DataCloud, como *PaaS Orquestrator* [36] y otros componentes de autenticación y autorización y en el futuro, tratará de tener en cuenta varias integraciones sobre herramientas de *Big Data*. Esta solución tiene en cuenta el aprovisionamiento de recursos y su instalación: se iniciarán máquinas virtuales y se configurarán de manera automática usando TOSCA y Ansible [18]. Ansible es una herramienta diseñada para la configuración automática de aplicaciones permitiendo descargar y configurar cada uno de los componentes de nuestro caso de uso de manera automática a través de roles. Los roles, en Ansible, son capaces de crear tareas y subtareas para instalar y configurar la aplicación de manera organizada y automatizada, como si fuera un proceso encadenado. Además los roles pueden ser compartidos por más de un usuario.

Hay que remarcar que esta solución se encuentra en una fase de *testing* y este trabajo fin de máster ha formado parte de esas pruebas. Toda la parte de integración de la solución la he realizado con ayuda de los desarrolladores de la misma una vez se comenzaron a producir los primeros problemas explicados en la Sección 3.10.

En la Figura 3.1 se muestra gráficamente en qué consiste la solución. El primer

paso es la autenticación del usuario en el sistema para poder acceder, posteriormente, a los recursos. Una vez se autentique y a su vez se definan los parámetros y recursos del entorno a desplegar en el template de TOSCA, se lanzará el despliegue a través del servicio de orquestación, Heat. Inicialmente se desplegó sobre la solución *PaaS Orchestrator* de INDIGO-DataCloud, pero en nuestro caso, al ejecutarse sobre una infraestructura determinada, se realizará a través del servicio Heat de OpenStack. Una vez se haya creado todo el entorno, es necesario incluirlo en la *Global Pool* de *HTCondor* para el proyecto CMS. *HTCondor Global Pool* es una infraestrutura de cómputo compartida por todos los usuarios de CMS que permite lanzar *jobs* del experimento en cualquiera de los clústers que pertenezcan a la cola. Para ello, los agentes de Mesos, que se explicará en la Sección 3.1.1, se conectarán al servicio *Token Translation Service*(TTS) y generarán las credenciales necesarias para ejecutar los análisis que se enviarán, posteriormente a la *pool* de *HTCondor* para que sean verificadas. Una vez se den permisos para unirse a la *Global pool*, los usuarios podrán escoger nuestro clúster para lanzar las simulaciones y que estas se ejecuten en el entorno *cloud* que vamos a desplegar.

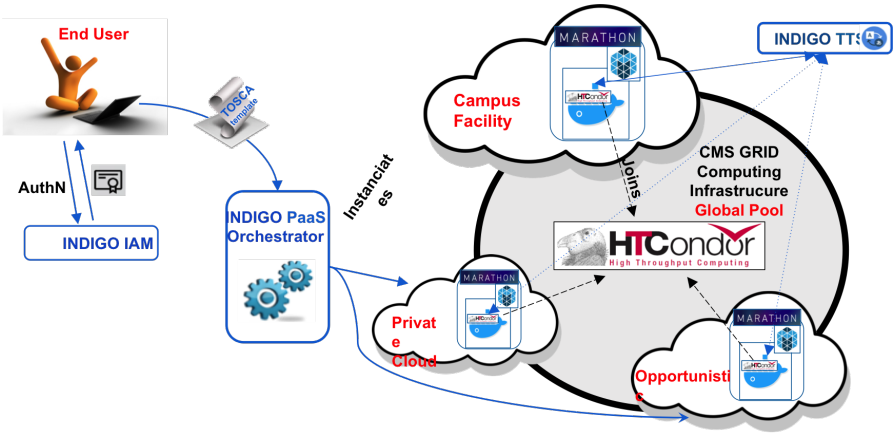

Figura 3.1: Arquitectura de DODAS: *Dynamic On Demand Analysis Service*. Fuente [24].

Para la definición del clúster se ha elegido Mesos, descrito en la Sección 3.1.1. Una vez se complete el despliegue del clúster, estarán listos todos los servicios requeridos para ejecutar una simulación CMS (*worker nodes*, *squid caches*, *proxy cache services*, etc). Estos componentes se irán viendo a medida que se detalle cómo se ha realizado el proceso.

Como se explicó en el Capítulo 1 el objetivo principal es que todo el proceso de automatización incluya el despliegue y la configuración del *software* necesario para el experimento y todo se realice de manera transparente de cara al usuario.

3.1.1. Componentes de la solución DODAS

Antes de crear el *template* de TOSCA especificando la topología del servicio, es necesario definir los diferentes componentes de la solución. Para crear el entorno se ha elegido la herramienta de Mesos, por lo que la solución final es desplegar un clúster de Mesos cuyos nodos ejecutarán las aplicaciones que se lanzan a través del *framework* Marathon. En los siguientes apartados iré introduciendo y explicando cada uno de los

componentes de la aplicación.

Apache Mesos

Apache Mesos [19] es una herramienta que gestiona recursos de cómputo (cpu, memoria, almacenamiento, etc) y los particiona de manera eficiente para cada una de las aplicaciones o *frameworks*. Mesos soporta una lista limitada de aplicaciones, resumidas en [20].

Mesos tiene la ventaja de ser un sistema que soporta las diferentes necesidades de cada aplicación teniendo como principales características la escalabilidad y la eficiencia. Primero, el sistema es capaz de escalar a más de 50.000 nodos y correr cientos de *jobs* con millones de tareas. Segundo, ya que cada *framework* tiene unas necesidades diferentes, Mesos es tolerante a fallos ofreciendo una gran disponibilidad. Para ser tolerante a fallos, Mesos utiliza otro sistema llamado Zookeeper explicado más adelante. Por tanto, Mesos implementa un planificador central que toma los requisitos de las aplicaciones como entrada para después, planificar cómo ejecutar cada tarea.

En la siguiente lista se resumen las pricipales ventajas de Mesos:

- Escalabilidad: despliegue en más de 50.000 nodos (emulados).
- Aislamiento de los recursos para la ejecución de las tareas a través de contenedores.
- Planificación eficiente de recursos.
- Alta disponibilidad a través de Apache Zookeeper.
- Interfaz Web para monitorizar el estado del sistema.

Arquitectura de Mesos

Mesos está compuesto por tres componentes básicos como se ve en la Figura 3.2: ***masters***, ***slaves*** o agentes y ***frameworks***. El nodo *master* gestiona los *slaves* y los nodos *frameworks* se encargan de enviar tareas para que se ejecuten en estos últimos. Estos *frameworks* son las aplicaciones de Mesos que van a ejecutar sus trabajos sobre el entorno. Están compuestos por un *scheduler* que se registra en el máster para recibir y enviar los *resource offers*, y uno o más *executors* que ejecutan tareas en los agentes. En las dos siguientes secciones se explicarán dos ejemplos de *frameworks*: Marathon y Chronos.

El máster gestiona la planificación usando *resource offers*. Cada uno de estos *resource offer* contiene una lista de los recursos libres que contienen los nodos agentes y así, el máster puede decidir cuántos recursos puede ofrecerle al *framework* de la aplicación de acuerdo con una política de planificación. Esta política puede ser o por prioridades, o compartiendo equitativamente los recursos entre las aplicaciones. Mesos soporta un gran número de políticas pero suele dejar a las organizaciones definir sus propias políticas.

Mientras el máster decide los recursos que ofrecerá a las aplicaciones, los *schedulers* seleccionan cuáles de los recursos que se han ofrecido van a querer usar. Una vez que el *framework* elige los recursos y los acepta, le pasa a Mesos una descripción de las tareas que quiere ejecutar. Las tareas, se pueden definir, como la unidad de trabajo enviada por el *framework* que es ejecutada en el nodo *slave*. Ejemplos de tareas pueden ser un *script*, un comando en *bash* o una *query* de SQL. Una vez que las tareas se han

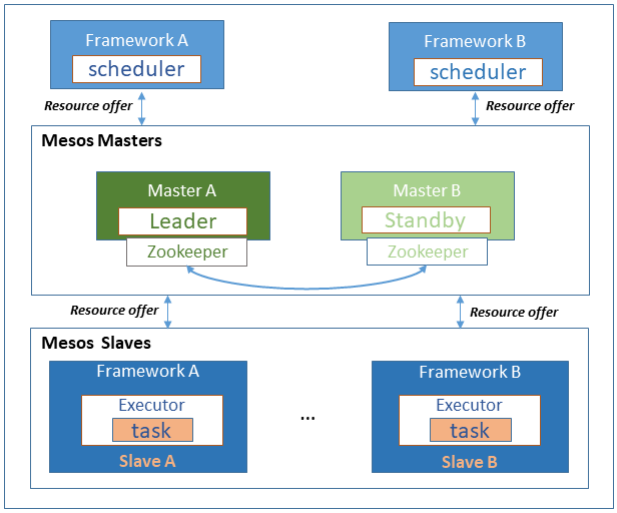

Figura 3.2: Arquitectura Apache Mesos.

completado y una parte de los recursos vuelven a quedar libres, se vuelve a repetir todo el proceso comenzando con el envío de un *resource offer* desde los agentes al máster especificando los recursos disponibles para que vuelva a enviar tareas.

Zookeeper

Zookeeper es un proyecto de software libre de Apache que provee un servicio centralizado para coordinar la información de todos los servicios y recursos de un sistema distribuido para poder, posteriormente, sincronizarlos y crear grupos de procesos de manera segura y eficiente.

Zookeeper proporciona a Mesos alta disponibilidad ya que coordina a los másteres de manera que pueda elegir qué nodo máster es el lider y cuáles se configuran en modo *standby*. Una vez que Zookeeper detecta que el líder falla, elige un nuevo máster como líder para que los *frameworks* y los *slaves* se puedan conectar a él y actualizar su estado.

En la Figura 3.2 de arriba, se muestra una arquitectura sencilla de Mesos para no recargar la figura con más elementos. Para hacer posible una alta disponibilidad se requiere que existan, al menos, tres nodos máster y así poder mantener la disponibilidad si uno de los máster falla. Aun así, para un entorno en producción sería recomendado tener tres nodos máster y seguir manteniendo el concepto principal de Mesos si dos de los nodos fallan.

Marathon

Marathon es una plataforma de orquestación de contenedores que juega el papel de *framework* en Mesos. Está diseñado para lanzar aplicaciones en Mesos. A continuación se resumen sus características más importantes para conseguir simplificar la ejecución de aplicaciones en un clúster:

- **Alta disponibilidad**: Se ejecuta en un clúster de Mesos dónde siempre existe un nodo máster que actúa como líder gracias a la ejecución del servicio Zookeeper.

- **Ejecución de múltiples tipos de contenedores**: soporta, tanto los contenedores propios de Mesos (usando *cgroups*) como Docker.
- **Optimización del sistema**: Existe un control pleno del sistema donde las aplicaciones se ejecutan de la manera más optimizada posible: **tolerancia a fallos** donde una tarea se propaga en múltiples nodos, o **localidad**, donde las tareas corren en un mismo nodo.
- **Balanceo de carga y control continuo del sistema**: chequeo del correcto funcionamiento de todos los recursos y servicios del sistema.
- **REST API**: Mejorar la integración, escalabilidad y el envío de tareas a los nodos.

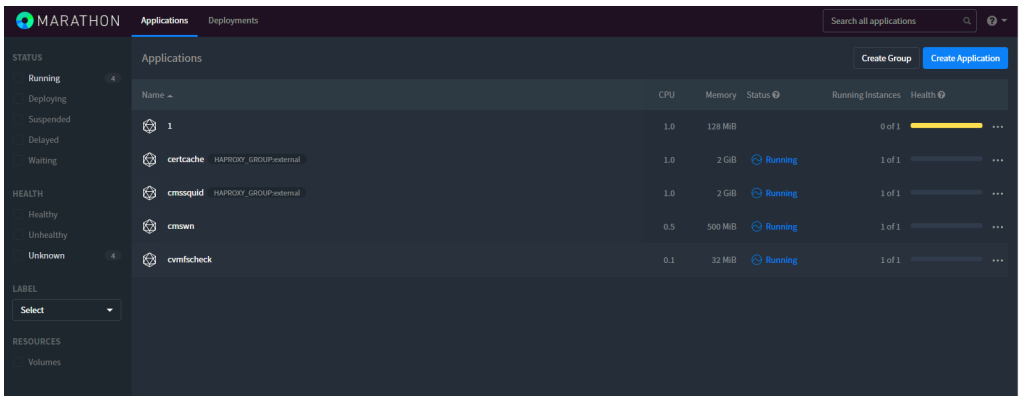

Figura 3.3: Interfaz Web de Marathon.

Como se muestra en la Figura 3.3, Marathon puede gestionar otras muchas aplicaciones, como es el caso de Chronos que se explicará en detalle a continuación, así como lanzar comandos directamente en *bash*. Marathon puede arrancar, pausar o detener las aplicaciones y está diseñado para lanzar aplicaciones largas y asegurar que estás continúen ejecutándose incluso en el caso en el que los nodos *slaves* donde se están ajecutando hayan fallado.

Chronos

Chronos es una aplicación que se ejecuta sobre el clúster de Mesos y que permite la orquestación de las tareas. Se asemeja al comando **cron** de Unix. Chronos se define como un planificador distribuido y tolerante a fallos para Mesos que soporta el concepto de *executor* así como la ejecución de tareas vía línea de comandos.

Esta herramienta complementa a Marathon al proveer otra manera de ejecutar aplicaciones de acuerdo a una planificación o a la finalización de otro trabajo. También es capaz de planificar *jobs* en los nodos agentes de Mesos y proveer estadísticas de fallos de los mismos.

3.2. Definición del clúster de Mesos en DODAS

Mesos, como se acaba de definir en la sección anterior, es una herramienta que gestiona un clúster ofreciendo aislamiento de los recursos y compartiendo y ejecutando

aplicaciones de manera distribuida asegurando escalabilidad y eficiencia. La arquitectura de Mesos viene definida por tres nodos principales: máster, *load balancer* y agente. En DODAS, Docker se aloja sobre Mesos para instalar cada uno de los componentes de la solución.

A continuación, se muestra la arquitectura del clúster de Mesos según los requisitos necesarios por la aplicación DODAS que se va a integrar.

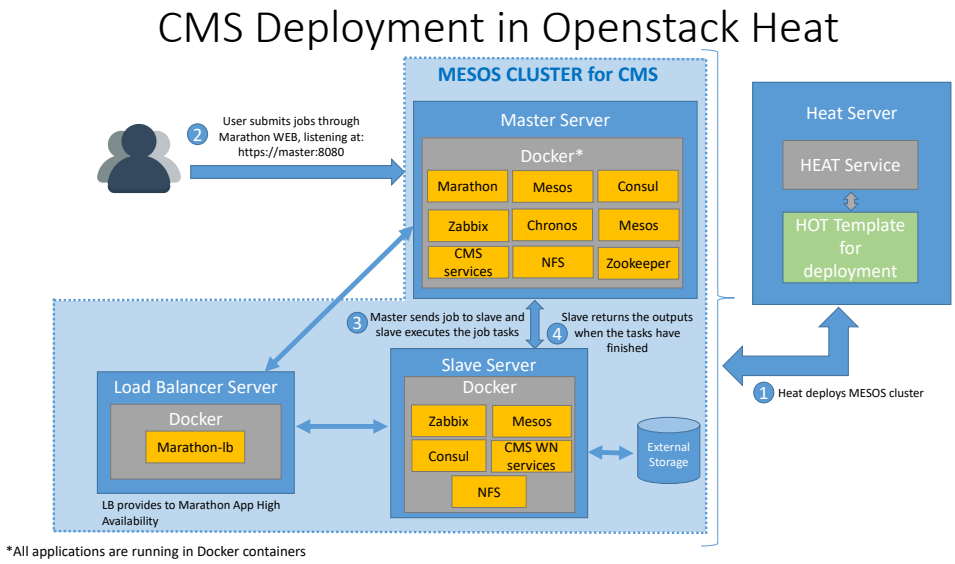

Figura 3.4: Arquitectura del cluster de Mesos.

Como vemos en la figura 3.4 todas las aplicaciones que necesitamos para desplegar la arquitectura son ejecutados a través de contenedores Docker, explicado en la Sección 2.4 del capítulo anterior, en cada uno de los nodos. Cada uno de ellos se encapsula en un *rol* para que cada componente pueda ser descargado y posteriormente configurado a través de Ansible.

Las aplicaciones que se ejecutan en cada nodo son las siguientes:

- **MÁSTER**: Marathon, Mesos, Consul, Zabbix, Chronos, Zookeeper, NFS y servicios necesarios para las aplicaciones de CMS.
- ***LOAD BALANCER***: Marathon-lb.
- **AGENTE**: Mesos, Zabbix, Consul, NFS y servicios de CMS para los *worker nodes*.

Como el gran peso de la solución lo llevan los componentes ya descritos anteriormente en este capítulo y en el anterior, se resumirán a continuación, aquellos que no se vieron anteriormente y cuál es la funcionalidad que ofrecen en nuestro caso de uso.

- **Consul**: Herramienta o servicio cuyo principal objetivo es la detección de otros servicios que se encuentran en la misma red, por ejemplo DNS o HTTP.
- **Zabbix**: Sistema de monitorización diseñado para controlar el estado de los recursos de cómputo, red o almacenamiento. Al estar en una etapa de testeo, esta herramienta no se va a utilizar, de momento y no se va a comprobar su

funcionalidad. En un futuro servirá de ayuda para registrar los servidores en la herramienta y monitorizar los servicios de manera centralizada. Mesos también muestra información sobre las máquinas que están levantadas o apagadas, pero con Zabbix conseguimos detectar porqué ocurre además de mostrar errores en otros recursos de la infraestructura que necesitan dichos componentes.

- **NFS**: *Network File System*. Es un protocolo utilizado en sistemas distribuidos para acceder a ficheros remotos como si se tratara de locales. En DODAS, permitirá almacenar los datos de la aplicación. El nodo agente se comunica con el sistema de almacenamiento, a través de NFS, para que las aplicaciones guarden sus datos.

- **Marathon-lb**: Herramienta basada en *HAProxy*. *HAProxy* es un balanceador de carga y un servidor proxy que ofrece alta disponibilidad propagando las peticiones TCP y HTTP a través de múltiples servidores. En la solución se encarga de balancear los servicios de Apache Mesos, más concretamente, balancea la carga entre los contenedores Docker de los nodos agentes. Es decir, estos contenedores se ejecutan en nodos agentes y puertos aleatorios y ésto hace complicado que se pueda trabajar con ellos a no ser que exista un punto de entrada que envíe las aplicaciones de los usuarios de una manera más eficiente. Este punto de entrada lo proporciona *marathon-lb* como se ve en la figura 3.5.

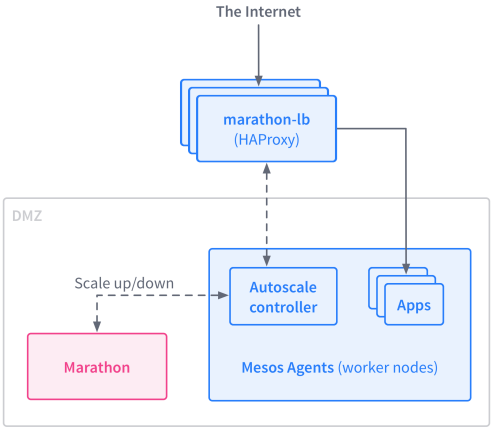

Figura 3.5: Arquitectura Marathon-lb. Fuente DC/OS.

Marathon-lb es un *script* que se conecta con Marathon y va actualizando la configuración de *HAProxy*. Por cada aplicación que lance el usuario desde Marathon y se ejecute en Mesos, Marathon-lb hará la asociación entre el puerto de entrada que le hayamos configurado y los respectivos puertos que se configuran en los contenedores.

3.3. Definición del *template* TOSCA

Una vez descritos los componentes de DODAS que se instalarán en el *cloud* del IFCA, se debe definir la topología del servicio a través del estándar abierto llamado TOSCA descrito detalladamente en la Sección 2.3.2. Por tanto, el primer paso es crear un fichero basado en TOSCA donde he definido toda la topología de la solución adaptándola a la infraestructura local.

INDIGO-DataCloud ha adoptado TOSCA como lenguaje para la solución debido a su **interoperabilidad** ya que es capaz de definir aplicaciones, servicios, plataformas, datos e infraestructuras *cloud* basados en sus políticas, requisitos y capacidades. Además, su principal ventaja radica en los conceptos de **portabilidad** y **automatización** de tareas ya que hace posible que el entorno se pueda desplegar independientemente de la plataforma sobre la que se desarrolle.

Para crear el fichero en TOSCA, cada tipo de componente viene predefinido en un template YAML de INDIGO-DataCloud donde se han ido añadiendo nuevos tipos de componentes necesarios para el proyecto.

Anteriormente, en la Sección 2.3.2 se explicó la estructura del *template* a través de varios ejemplos, por lo que a la hora de definir cada parte, se da por hecho que ya se entiende la función de cada uno. En el siguiente enlace de git [29] se puede consultar el *template* completo modificado para ser implantado en la infraestructura *cloud* del IFCA.

En primer lugar, se incluye el *template* de INDIGO-DataCloud donde se definen los tipos de cada componente que se irán implementando. Este *template* también se basa en TOSCA y su objetivo es extender los tipos definidos por defecto en *TOSCA Simple Profile in YAML Version 1.0*. Para este caso, se usa el parámetro `imports`.

A partir del apartado comenzado por `topology_template` se crea toda la definición de nuestro entorno dividida en tres partes: `inputs`, `node_templates` y `outputs`.

Para explicar los parámetros de entrada (`inputs`) los dividiré según su propósito para la aplicación. Estas variables deben ser modificadas por el desarrollador para adaptarlo a la infraestructura local. Los tres primeros que se muestran en *Listing* 3.1 corresponden a los parámetros de autenticación que veremos en la Sección 3.4.

Listing 3.1: Definición de los parámetros de autenticación en TOSCA.

```
iam_token:
  type: string
  default: "my_token"

iam_client_id:
  type: string
  default: "myclient_id"

iam_client_secret:
  type: string
  default: "myclient_secret"
```

Los siguientes corresponden a parámetros requeridos para el experimento CMS. Son necesarios a la hora de que el usuario final pueda lanzar sus simulaciones dentro del proyecto de CMS y en la infraestructura del IFCA. Todos ellos comienzan por "`cms_`" y se irán explicando a continuación:

- **cms_local_site**: "`T3_ES_Opportunistic_IFCA`". Es un identificador que se utiliza en la cola de nodos de *HTCondor* de CMS para que el usuario final envíe los *jobs* a un determinado clúster. Cada *local_site* es definido con el formato "`TX_XX_Opportunistic_XXXX`". El primer carácter es el tipo de *tier* del clúster, el siguiente, el país de procedencia y los últimos, el nombre del centro donde se encuentra instalado.

- **cms_stageoutsite**: Opcional. "`T3_ES_Opportunistic_IFCA`". Se define de la misma forma que el anterior, pero se utiliza para obtener los resultados de la simulación. En este paso se suele verificar que los resultados de la simulación son correctos.
- **cms_stageoutserver**: Opcional. `srmfe01.ifca.es`. Si el parámetro anterior definía el *site*, en este caso, es el servidor donde se reciben los resultados finales del trabajo.
- **cms_staoutprefix**: `srm://srmfe02.ifca.es:8444/srm/managerv2?SFN=/cmsdisk/`.
- **cms_*_fallback**: Opcional. "DUMMY". No se van a utilizar para nuestro caso de uso, por lo que se les deja por defecto sin modificar.
- **cms_input_path**: Opcional. No es necesario para nuestro caso de uso por lo que se deja por defecto.
- **cms_input_protocol**: Opcional. `xrootd`. Servicio de análisis de los datos de entrada para CMS. No importa en qué infraestructura *Grid* estén localizados ya que se analizan sin tener que ser descargados al sistema de almacenamiento local.
- **cms_squid_image**: "`spiga/frontiersquidv1`". Imagen de Docker para el servicio de *Squid*. Es un servidor *proxy* que se utiliza para incrementar el rendimiento de los accesos y consultas que se realizan a servidores web. Guarda en caché las peticiones a servidores web y DNS.
- **cms_wn_image**: "`spiga/cmswn:newccb`". Imagen de Docker para los servicios de los *worker nodes* que serán desplegados en los nodos agentes.
- **cms_proxycache_image**: "`spiga/ttscache`". Imagen de Docker para el servicio *Proxy* Caché. Es un servidor conectado entre el cliente y otra red que actúa como protección separando las dos redes y como zona de caché para acelerar el acceso a páginas web o poder restringir el accesso a contenidos.

Las dos siguientes variables pertenecen al servicio de INDIGO-DataCloud, **Onedata**. Este servicio tiene como objetivo ofrecer un sistema de almacenamiento de datos de manera compartida entre usuarios de los distintos centros que componen el proyecto. Permite montar y desmontar dinámicamente un directorio según lo requiera el usuario. Para la utilización del servicio es necesario que el cliente esté correctamente autenticado a través de un ***token***. De momento, se dejarán con su valor por defecto ya que no interferirán en el objetivo inicial de este proyecto.

`Monitordb_ip` se usa para conectarse al servicio de monitorización del entorno desplegado. Una vez que se ponga en producción, sí será necesario para controlar el entorno. En un futuro próximo se intentará integrar con el sistema de monitorización local pero de momento se utilizará el sistema de monitorización instalado en el INFN ya que este apartado no entra dentro de las pruebas realizadas en este trabajo.

A continuación, en *Listing* 3.2 se define la parte principal del *template*. Esta parte comienza con la variable `node_templates` y, a partir de aquí, se irán definiendo cada uno de los nodos que componen el clúster. Para no replicar información, aquí solo se mostrará la configuración del nodo máster y se podrá consultar el Anexo B donde se

detalla todo el *template*. Como ya se explicó anteriormente, el clúster se compone de tres nodos principales: máster, agente y balanceador. Para la definición de cada nodo es necesario dividirlo en dos pasos:

1. Creación del nodo de cómputo con los requerimientos que necesita para poder arrancar sus aplicaciones.

   Listing 3.2: Definición del nodo máster de Mesos a través de TOSCA (1).

```
node_templates:

  mesos-master-server:
      type: tosca.nodes.indigo.Compute
      capabilities:
        endpoint:
          properties:
            network_name: PUBLIC
            dns_name: mesosserverpublic
            ports:
              mesos_port:
                protocol: tcp
                source: 5050
              marathon_port:
                protocol: tcp
                source: 8080
        scalable:
          properties:
            count: 1
        host:
          properties:
            num_cpus: 4
            mem_size: 8 GB
        os:
          properties:
            image: ubuntu16-software-config-ifca
```

   En el caso del máster, solo necesitamos un servidor, como mínimo, para poder ejecutar nuestra aplicación, aunque según mi criterio, deberían ser mínimo dos para sustituir el máster primario por el secundario en el momento en el que uno de ellos falle. Para obtener la escalabilidad que proporciona Mesos, los nodos agentes deben ser como mínimo tres para poder ejecutar las simulaciones que se envíen sin perder rendimiento. Por ejemplo, en el caso en el que haya más de un usuario ejecutando una simulación con los mismos requerimientos con los que cuenta el servidor. Todo esto se define a partir de la variable `scalable`.
   Además, para el caso que se muestra en Listing 3.2, es necesario asignar una IP pública con el nombre de dominio "`mesosserverpublic`" al nodo máster y abrir los puertos 5050 y 8080 para las aplicaciones de Mesos y Marathon, respectivamente, con el objetivo de que los usuarios puedan acceder a ellas una vez que esté lista la aplicación. Esto se define dentro del parámetro `network_name`

en `properties`, que a su vez está dentro de `endpoint`. Lo mismo ocurre para el balanceador, ya que también, se le asigna una IP pública con el nombre de dominio "`mesoslb`".
Cada uno de los tres nodos son máquinas virtuales que se arrancan con la imagen de Linux Ubuntu 16.04 instalada, previamente, instalada a través del servicio Glance de OpenStack. La creación de esta imagen se explicará en la Sección 3.5 ya que se necesita añadir cierta funcionalidad a la imagen de Ubuntu que se instala por defecto, para utilizarla en *Heat*.

2. Configuración del nodo asignándole parámetros que son posteriormente utilizados a la hora de instalar las aplicaciones a través de Ansible.

   Listing 3.3: Definición del nodo máster de Mesos a través de TOSCA (2).

```
mesos_master:
      type: tosca.nodes.indigo.MesosMaster
      properties:
        marathon_password: test_pass
        chronos_password: test_pass
        mesos_masters_list: { get_attribute:
         [ mesos-master-server, private_address ]}
      requirements:
        - host: mesos-master-server
```

   En el *template* donde se definen los tipos de componentes existe un nodo del tipo `MesosMaster`, así como `MesosSlave` y `MesosLoadBalancer`. En base a su tipo es obligatorio asignar ciertos parámetros como los que se muestran en *Listing*3.3 : `marathon_password`, `chronos_password` y `mesos_masters_list`. Los dos primeros son necesarios para que el administrador se conecte a las web de Marathon y pueda utilizar el servicio de Chronos. La última propiedad asigna una ip privada al nodo. Además, se definirá con `host` el nodo al que se pasarán las anteriores propiedades.

Para establecer los parámetros correspondientes al lanzamiento de aplicaciones para CMS se deben crear otros dos apartados correspondientes a los servicios del nodo máster y de los nodos agentes, que realizarán las funciones de *worker nodes*(WN). Los *worker nodes* en CMS son los encargados de ejecutar la simulación que se envía desde los *frontends*, en este caso, el máster. Estos dos nodos solo configuran los parámetros que dan la funcionalidad de CMS al máster y a los agentes antes de ser arrancados a través de los servicios de Heat explicados en la Sección 3.5. Los parámetros se definirán dentro de la variable `properties` y corresponden a los explicados anteriormente dentro del apartado `inputs`.

- ***cms_services***: Servicios del nodo máster. Se definen con el tipo ***CmsServices*** de INDIGO-DataCloud. Además de los parámetros de autenticación correspondientes al IAM, *Onedata* y *Marathon*, se asignan los servidores *Squid* y *Proxy Caché* y el local site de CMS.

- ***cms_wn***: Servicios de los *worker nodes* correspondientes a los nodos agentes. Se define con el tipo ***CmsWnConfig***. Como se trata de los servidores que ejecutarán las simulaciones se les tiene que pasar todas las variables de entrada para CMS explicadas anteriormente.

Una vez definido el entorno en el *template* de TOSCA se debe traducir a un lenguaje entendible por el servicio de orquestación de OpenStack, **Heat**. El servicio de orquestación de OpenStack solo es capaz de entender y desplegar aquellas implementaciones definidas en el lenguaje HOT. Antes de explicar cómo realizar la traducción, en la siguiente sección se detallará cómo se realiza la autenticación y autorización de los usuarios para poder acceder a los recursos desplegados antes de lanzar sus trabajos.

3.4. Autenticación y Autorización

El proceso de Autenticación y Autorización se realiza a través del servicio IAM de INDIGO-DataCloud. Este servicio se basa en el estándar *OpenID Connect* [30] que proporciona una capa de identificación sobre el protocolo *Oauth 2.0* [25] de manera que siga el proceso que se muestra en la Figura 3.6. Todo este proceso se detalla en profundidad en [25].

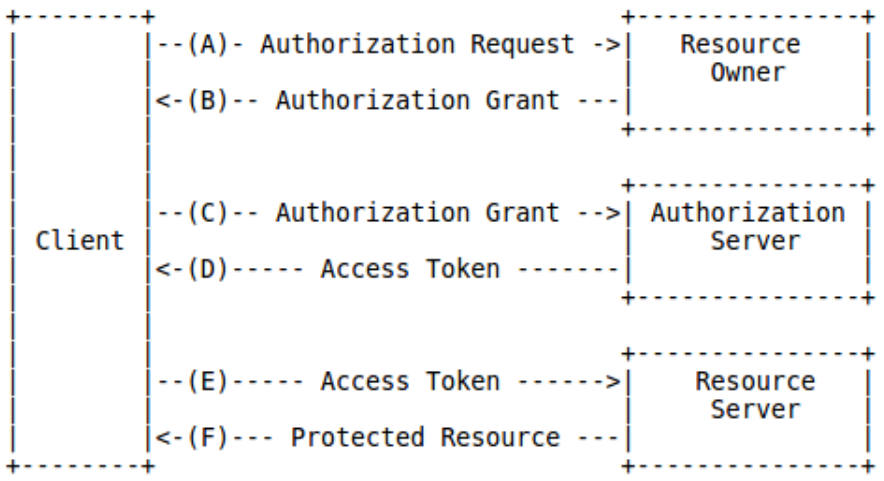

Figura 3.6: *Workflow* Protocolo *Oauth 2.0*. Fuente [25].

3.4.1. Identity and Access Management(IAM)

El servicio de INDIGO-DataCloud IAM (*Identity and Access Management*) provee una nueva capa de autenticación e identificación para que el usuario pueda acceder a los recursos y servicios distribuidos. Soporta mecanismos de autenticación de INDIGO-DataCloud AAI (*Authentication and Authorization Infrastructure*). Sus principales características son las siguientes:

- **Autenticación**: Soporta mecanismos de autenticación definidos en INDIGO-DataCloud AAI (SAML, x.509, OpenID connect).
- **Gestión de Sesiones**: Provee gestión de sesiones. Una sesión es usada para proveer *Single Sign-On* (SSO) y funcionalidad *logout* para aplicaciones y servicios.
- **Inscripción**: Provee mecanismos de registro para usuarios que desean unirse a grupos u organizaciones.

- **Gestión de identidad**: Provee servicios de gestión de grupos, asignación de atributos a los administradores de los grupos y la creación de múltiples identidades para una única identidad de INDIGO-DataCloud.

En la Figura 3.7 se muestra el proceso de autenticación de nuestro caso de uso.

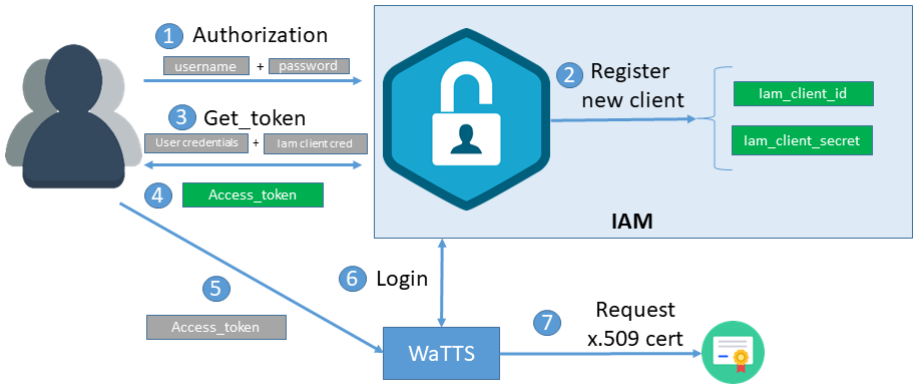

Figura 3.7: Proceso AAI de DODAS.

Para comenzar el proceso de autenticación y autorización debemos acceder al servicio IAM [22]. Para ello, debemos crear previamente una cuenta en dicho servicio. Una vez hayamos creado la cuenta podremos iniciar sesión con las credenciales proporcionadas por el servicio de autenticación, nombre de usuario y contraseña, a través de la página de login del servicio IAM [22] como se ve en la Figura 3.8. Dentro de nuestro perfil es necesario registrar un nuevo cliente que se usará para obtener un *token*, y así, posteriormente poder acceder a los recursos de cómputo. Al registrar el nuevo cliente se obtendrán un *id* y un *secret*. Con ese *id* y ese *secret* más nuestras credenciales de IAM podremos crear un *access token* como se explica en el siguiente apartado.

Los ***access token*** son credenciales usadas para acceder a unos recursos protegidos. En otras palabras, un *access token* se representa como un string que se envía al usuario para que se autorice en los servicios. La ventaja de estos *tokens* es que ofrecen una mayor seguridad al servicio debido a que la duración de acceso es limitada y son actualizados a través de los llamados *refresh token* cada cierto período de tiempo.

Para obtener el *token* es necesario utilizar un *OpenID Client* como el que se detallará a continuación.

Obtener Token

Antes de lanzar el despliegue debemos definir en el *template* de TOSCA las credenciales correspondientes a la autenticación y autorización del sistema. Para ello debemos utilizar un *OpenID client* como el que se crea en el *Listing* 3.4.

Mediante el uso de ***Curl*** es posible conectarse al servidor de IAM y realizar una petición para obtener el *token*. Es necesario pasarle a *Curl* como parámetros las credenciales del cliente, las credenciales del usuario, un *scope* que define los tipos de datos accesibles para las aplicaciones (email, *profile*, *openid*, etc) y el *endpoint* de INDIGO-DataCloud donde se conecta el cliente para recibir el *token*. El comando *jq* es una especie de filtro ya que toma una entrada para producir una salida en base al parámetro que pida el usuario, *access token* en nuestro caso.

Listing 3.4: get token vía script.

```
#!/bin/bash

IAM_CLIENT_ID=CHANGEME
IAM_CLIENT_SECRET=CHANGEME
IAM_USER=

IAM_CLIENT_ID=${IAM_CLIENT_ID:-iam-client}
IAM_CLIENT_SECRET=${IAM_CLIENT_SECRET}

if [[ -z "${IAM_CLIENT_SECRET}" ]]; then
  echo "Please provide a client secret setting
  the IAM_CLIENT_SECRET env variable."
  exit 1;
fi

IAM_USER="CHANGEME"
IAM_PASSWORD="CHANGEME"

result=$(curl -s -L  \
  -d client\_id=${IAM_CLIENT_ID} \
  -d client_secret=\${IAM_CLIENT_SECRET} \
  -d grant_type=password  \
  -d username=${IAM_USER} \
  -d password=${IAM_PASSWORD} \
  -d scope="openid profile email offline_access" \
  \${IAM_ENDPOINT:
  -https://iam-test.indigo-datacloud.eu/token})

if [[ $? != 0 ]]; then
  echo "Error!"
  echo \$result
  exit 1
fi

echo $result

access_token=$(echo $result) #| jq -r .access_token)

echo ${access_token}
```

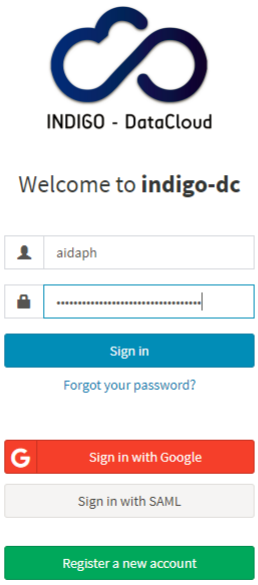

Figura 3.8: INDIGO-DataCloud IAM *Login*.

En vez de usar el script, también podemos obtener el token a través de la línea de comandos ejecutando el comando especificado en *Listing* 3.5:.

Listing 3.5: get token vía bash.

```
$ curl -s -L -d  \
       client_id="XXXX" -d client_secret="XXXX"  \
       -d grant_type=password -d username="CHANGEME" \
       -d password="CHANGEME" -d scope="openid profile
        email offline_access"  \
        https://iam-test.indigo-datacloud.eu/token
```

En cualquiera de los dos casos, nos devolverá un *access token* que se pasa en el parámetro `iam_token` del *Listing* 3.1. Los otros dos parámetros del *template*, `iam_client_id` y `iam_client_secret` corresponden a las credenciales, *id* y *secret*, del cliente que registramos anteriormente en el servicio de IAM.

3.4.2. Token Translation Service (TTS)

Para finalizar todo el proceso de autenticación que se muestra en la Figura 3.7 debemos conectarnos al **servicio TTS** (*Token Translation Service*). Este servicio es el encargado de recibir un *token* y devolvernos un **certificado x.509** que permitirá acceder a la *Global Pool de HTCondor*. El *token* es enviado por los *worker nodes* una vez están listos y ejecutándose (es por eso por lo que se pasa el *token* directamente en el *script*). La firma del certificado será comprobada en *HTCondor* para así certificar que el clúster es válido para ser accesible por los usuarios. Para continuar en la misma línea, nos conectaremos al servicio TTS de INDIGO-DataCloud, WaTTS [23] y nos loguearemos seleccionando en la lista desplegable de *Identity Providers* (IDPs) el IAM como se ve en la Figura 3.9.

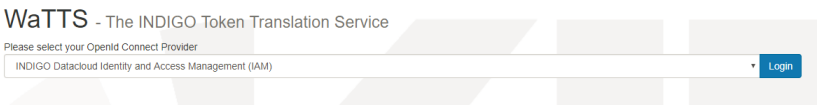

Figura 3.9: Servicio WaTTS - INDIGO *Token Translation Service*(TTS).

3.5. Creación imagen de Heat

Como se comentó anteriormente, para crear un nuevo servidor a través de Heat la imagen que se utilice para arrancar debe contar con una funcionalidad apropiada. Es decir, sobre la imagen por defecto que se usa normalmente para arrancar nuestros equipos deben estar pre-instalados otros componentes para poder llevar a cabo la configuración que instale y deje listas las aplicaciones del clúster que vayamos a desplegar.

Una imagen de Heat tiene que tener instaladas las siguientes herramientas:

1. **Cloud-init**: es una herramienta que inicializa una instancia de *cloud*. Una vez el servicio comienza a ejecutarse en el arranque de la instancia *cloud*, realiza lo siguiente:

   a) Configura el *hostname* del servidor

   b) Genera las llaves privadas de SSH.

   c) Añade las llaves en el fichero *.ssh/authorized_keys* del usuario para que pueda loguearse

   d) Monta los Sistema de Ficheros para que queden accesibles para el usuario.

2. **heat-cfntools**: Herramientas con los componentes necesarios para arrancar imágenes en Heat: *cfn-init*, *cfn-signal* y *cfn-hup*.

3. Tener conexión con un servidor de *metadata* que esté activo y corriendo. Suele tratarse de un servidor al que se accede a través de la dirección de enlace local 169.254.169.254.

4. La API de Heat activa en la máquina.

Si no queremos crear una imagen propia podemos descargarnos una de las imágenes preconfiguradas que ya existen a través de [37]. Están configuradas por el equipo de desarrolladores del servicio Heat. Si queremos crear una imagen propia podemos seguir esta guía [31] y subir la imagen al catálogo de imágenes del servicio Glance de OpenStack local.

3.6. Creación del template HOT

Una vez está definido el *template* de TOSCA y explicados cada uno de los parámetros nos disponemos a realizar la traducción a HOT. Para ello, se utiliza la API de *Heat Translator* [6], definido anteriormente en la Sección 2.3.2. El paquete de *Heat-translator* se descargará e instalará vía Git [6] y Python, respectivamente, siguiendo las instrucciones del fichero *README.md* del paquete. *Heat Translator* se ejecutará desde la línea de comandos según el comando de *Listing* 3.6.

Listing 3.6: Traducción de TOSCA a HOT vía heat-translator.

```
$ heat-translator --template-file=/my/path/tosca_dodas.yaml
--template-type=tosca --output-file=/my/path/hot_dodas.yaml
```

Con el parámetro `--template-file` se le pasa el *template* en TOSCA como entrada, `--template-tosca` es el tipo de fichero que se le pasa a Heat Translator, en este caso *tosca*, y `--output-file` es el *template* de salida en lenguaje HOT.

En la Sección 2.3.1 se explicaron detalladamente cuáles son los principales elementos del *template* de HOT. Después de tener varios problemas, explicados más detalladamente en la Sección 3.10, he tenido que adaptar el fichero inicial de HOT para conseguir un *template* final que se puede consultar en el correspondiente enlace de Git [33].

3.7. Componentes del *template* HOT

En base a los componentes del *template* de TOSCA que se explicaron en secciones anteriores se han añadido o modificado varios de ellos en el *template* de HOT. El resto de componentes que no se explican se dan por descritos en la Sección 3.3.

Parámetros de entrada

En la primera parte del *template*, que comienza con la variable ***parameters***, se han añadido los parámetros del *Listing* 3.7 correspondientes a la imagen de arranque, la red y subred, el *flavor* de cada instancia y el número de servidores que se lanzarán por cada nodo máster y agente de Mesos.

Listing 3.7: Parámetros de entrada template HOT.

```
server_image:
    type: string
    description: Image used to boot a server
    default: ubuntu16-software-config-ifca
network:
    type: string
    description: Network ID for the servers
subnetCMS:
    type: string
    description: Sub Network ID for the servers
ssh_key_name:
    type: string
    description: Name of keypair to access the servers
master_flavor:
    type: string
    description: flavor to use when booting the master
    default: cm4.xlarge
...
number_of_masters:
    type: number
    description: how many mesos masters to spawn
    default: 1
```

```
number_of_slaves:
    type: number
    description: how many mesos slaves to spawn
    default: 2
```

Recursos

En la sección principal que comienza con la variable `resources` se crean cada uno de los nodos de Mesos con su correspondiente configuración. Se divide en cuatro partes principales:

- ***Security Group***: Para filtrar el tráfico de red desde y hacia los servidores. Se definen a través del servicio de red Neutron de OpenStack bajo el tipo `OS::Neutron:: SecurityGroup`. Este *firewall* permitirá transferir el tráfico de los protocolos ICMP, TCP, UDP y SSH, además de abrir los puertos para las aplicaciones de *Mesos* y *Marathon*.
- ***Load Balancer***: Configuración del nodo *load balancer* de *Mesos*.
- **Máster**: Configuración del nodo máster de *Mesos*.
- **Agente**: Configuración de los nodos agentes de *Mesos*.

Para configurar cada uno de los nodos (`node*`) se deben seguir estos pasos:

1. `node*_setup_deployment`: Instalar nueva versión de *Ansible* en base a los problemas surgidos y explicados en 3.10 debido a la versión inicial de Ansible, y posteriormente, instalar los roles desde el repositorio de INDIGO-DataCloud vía *ansible-galaxy*. Este recurso es de tipo `OS::Heat::SoftwareDeployments` que permite asociar una configuración (`properties: config`) a varios servidores (`properties: servers`), donde esta configuración será desplegada. La configuración a instalar viene dada por el *script* definido en la variable `config` [32] dentro del recurso `ansible_setup_config`. En cada `SoftwareDeployment` se definen una serie de `input values` que se mapearán con los parámetros definidos dentro de la configuración de Ansible.

2. `node*_deployment`: configurar cada uno de los roles según la configuración que se le pasa en `node*_config` y los parámetros que se definen a partir de la variable `input values` del propio recurso. Estos valores se mapearán con los parámetros definidos en los roles de *Ansible*.

3. **loadbalancers**, **mesos_masters** y **mesos_slaves**: Definir cada uno de los servidores de *Mesos* a través del tipo `OS::Heat::ResourceGroup`. Para poder configurar cada servidor se le pasa: nombre, *key pair* para poder acceder, imagen de arranque, *flavor*, *network* y los *security groups* definidos anteriormente.

4. `cms_services_deployment`: Configuración de CMS para el nodo máster. Se le pasan los parámetros de entradas definidos y explicados en secciones anteriores.

5. `cms_wn_deployment`: Configuración de CMS para los nodos agentes. Se le pasan los parámetros de entradas definidos y explicados en secciones anteriores.

Outputs

En este apartado del *template* que comienza con la variable `outputs` se definen varios parámetros de salida correspondientes a las IPs de los nodos y a los *endpoints* donde conectarse para acceder a las aplicaciones:

- **Mesos**: "`http://host:5050`"
- **Marathon**: "`https://host:8443`"
- **Chronos**: "`http://host:4443`"

Se considera que se ha terminado el *deployment* una vez que estas variables se hayan definido y se le hayan pasado al servicio de Heat.

3.8. Creación del entorno de MESOS para CMS vía *Heat*

Para crear un nuevo *deployment*, denominado **stack** en Heat, he ejecutado la API del servicio a través del cliente de OpenStack como se indica en *Listing* 3.8.

Listing 3.8: creación del entorno de MESOS para CMS a través de Openstack.

```
$ openstack stack create mesos_cluster_cms
 -f yaml -t /my/path/hot_test.yaml
```

- `openstack`: Cliente de OpenStack.
- `stack create`: parámetros que se le pasan al cliente de Heat para crear el *stack*.
- `mesos_cluster_cms_ubuntu16`: nombre del *stack*
- `-f yaml`: Tipo de fichero que se pasa con el *template* para crear el despliegue.
- `-t hot_dodas.yaml`: Fichero con el *template* para crear el entorno. El mismo que creamos en la Sección 3.6.

Una vez lanzado el despliegue aparecerá un mensaje indicando que la creación del *stack* está en proceso. Si no es así debemos revisar el *template* de HOT para asegurarnos que todo es correcto como se indica en la Sección 3.10.

A continuación, se detallan los pasos que sigue Heat y el resto de clientes y herramientas de Openstack para crear el entorno:

1. *Heat-engine* genera un conjunto de datos que serán utilizados por *cloud-init*.
2. *Cloud-init* pregunta a Nova, servicio encargado del cómputo, para poder crear una instancia con los datos que se le pasan vía *cloud-init*.
3. El cliente de Nova selecciona un nodo donde arrancar la instancia con los requerimientos que le manda *cloud-init*.
4. Cuando arranca la instancia, se ejecuta el *script* de *cloud-init*. Este script realiza lo siguiente:
   a) Descarga los datos desde el servidor de metadatos.

b) Escribe las diferentes partes en el directorio */var/lib/cloud*.

c) Ejecuta todas las partes del *script* de *cloud-init*.

d) Ejecuta el fichero del usuario,`/var/lib/cloud/data/cfn-userdata`, ya que en algún punto debería llamar a *cfn-init*. Este `cfn-init` carga `/var/lib/cloud/data/cfn-init-data` que es una copia de los metadatos e instala paquetes, grupos de usuarios, usuarios, crea ficheros, etc.

Para que el *stack* se complete, el *deployment* debe enviar una llamada al servidor de Heat con los parámetros de salida del entorno. Estos parámetros son los *outputs* que se definieron en el *template* de HOT. Para mostrar la lista de *outputs* de un determinado *stack* hay que ejecutar los comandos que se muestran en *Listing* 3.9.

Listing 3.9: Stack output.

```
$ openstack stack output list mesos_cluster_cms
```

Como puede comprobarse en la Figura 3.10, el entorno **mesos_cluster_cms_ifca**, se ha desplegado correctamente ya que el estado del *stack* es ***CREATE_COMPLETE***. Hasta que este no finalice el estado será *CREATE_IN_PROGRESS*.

```
aidaph@heat:~$ openstack stack list
+--------------------------------------+----------------------+-----------------+----------------------+----------------------+
| ID                                   | Stack Name           | Stack Status    | Creation Time        | Updated Time         |
+--------------------------------------+----------------------+-----------------+----------------------+----------------------+
| 6b8b2427-5056-4574-83bf-59a3a13ba86f | mesos_cluster_cms_ifca | CREATE_COMPLETE | 2017-11-28T16:48:06Z | None                 |
```

Figura 3.10: Stack Completado.

3.9. Validación del entorno

Una vez el entorno se ha desplegado debemos conectarnos vía web a la aplicación de Mesos de nuestro clúster y comprobar que todos los componentes están listos para poder ejecutar aplicaciones según se muestra en la Figura 3.11. Para ello vamos a ejecutar una tarea de prueba muy sencilla sin llegar a ejecutar una simulación de CMS. En un futuro, se testeará con análisis reales. En este momento, cada nodo del nuevo entorno tiene asignadas unas IPs privadas dentro de la red interna. Para poder hacer accesible nuestro entorno y realizar las correspondientes comprobaciones se le asignará la IP 193.146.75.150 al nodo máster a través del cliente de OpenStack. Para acceder a Mesos debemos escribir en la barra de direcciones la IP, 193.146.75.150 y el puerto 5050 donde está instalado Mesos: `http://193.146.75.150:5050`. Una vez nos conectemos nos pedirá autenticarnos con las mismas credenciales introducidas en los parámetros `mesos_username` y `mesos_password` del *template* de HOT.

En la parte superior de la Figura 3.11 se muestran cuatro pestañas: *Mesos*, *frameworks*, *Agents* y *offers*. En la pestaña de ***Mesos*** se muestra la imagen de la figura con la página de inicio de la aplicación. A la izquierda de la imagen aparece información sobre el comportamiento actual del servicio Mesos. Primero se muestra el nombre del clúster, la IP del servidor, la versión de Mesos, así como la fecha de la versión y el comienzo del funcionamiento del clúster. Después se puede ver un *log* con todos los eventos ocurridos. Debajo, aparece información sobre los nodos agentes activos e inactivos así como de las tareas que han comenzado, finalizado, que se encuentran ejecutándose o fallaron. Por último, el total de recursos de cpu, memoria o disco usado

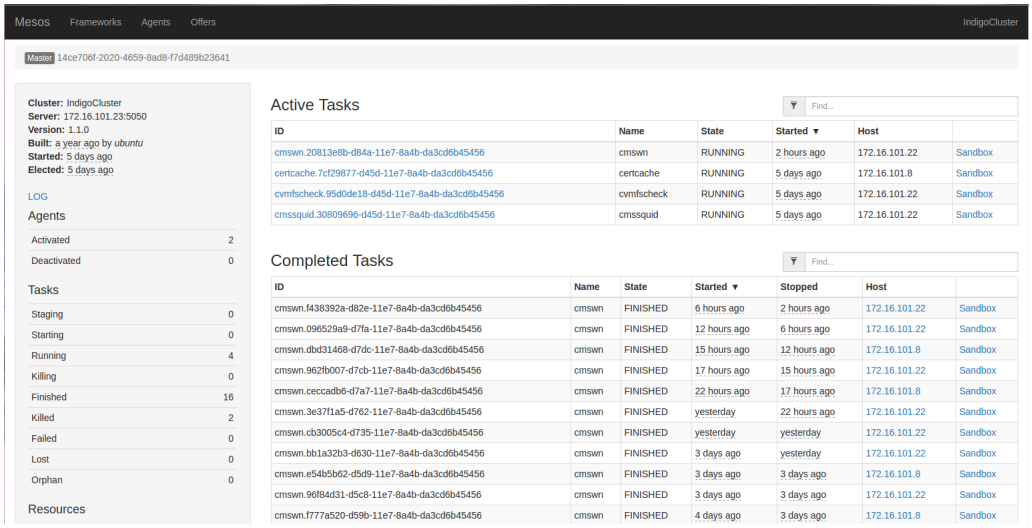

Figura 3.11: Web de Mesos del entorno DODAS desplegado en el IFCA.

o en estado *idle*. En la parte central de la imagen se muestran las tareas activas y completadas, así como, los nodos en los que se está ejecutando o se ha ejecutado cada tarea. En ***frameworks*** aparecen las aplicaciones activas o inactivas que enviarán sus trabajos a ejecutar a Mesos. En nuestro caso, como *framework* activo solo está configurado Marathon. En la pestaña ***agents*** se muestran los nodos agentes de Mesos, en nuestro caso únicamente dos. En ***offers*** aparecen los trabajos enviados por los *frameworks* y pendientes de ejecutarse en los nodos agentes.

Para comprobar el funcionamiento del servicio debemos ejecutar una nueva tarea desde la web de Marathon. Para conectarnos a ella, al igual que Mesos, accederemos a la aplicación de Marathon instalada en el puerto 8443: `http://193.146.75.150:8443` e introducir, también, las credenciales correspondientes de Marathon, `marathon_username` y `marathon_password` del *template* de HOT.

Como vimos en la Figura 3.3, ya existen 4 aplicaciones ejecutándose. Estas aplicaciones, correspondientes a CMS, son contenedores Docker que se generaron durante el despliegue y deben estar listas en los nodos agentes para poder ejecutarse una vez que los usuarios necesiten correr sus análisis de CMS.

Para ejecutar una nueva tarea, escogemos una sencilla como "Hola Mundo", y seleccionamos la pestaña ***Create Application*** para crear nuestra nueva aplicación con ID 1. Podemos asignar un número limitado de CPUs (1), memoria en MBs (128) e instancias(1) donde ejecutarla según se muestra en la Figura 3.12.

Como se ve en la Figura 3.13, en la aplicación de Mesos aparece la nueva tarea en estado completado después de ejecutarse en uno de los nodos agente disponibles con IP 172.16.101.22.

3.10. Problemas surgidos al crear el entorno vía Heat

Una vez traducido el *template* de TOSCA a HOT se comprobó con el comando que se muestra en la Figura 3.14 que al desplegar un nuevo *stack* nunca llegaba a completarse y no actualizaba su *status* de *CREATE_IN_PROGRESS* a *CREATE_COMPLETE*, como se observa en la Figura 3.14 con el *stack mesos_cluster_cms*.

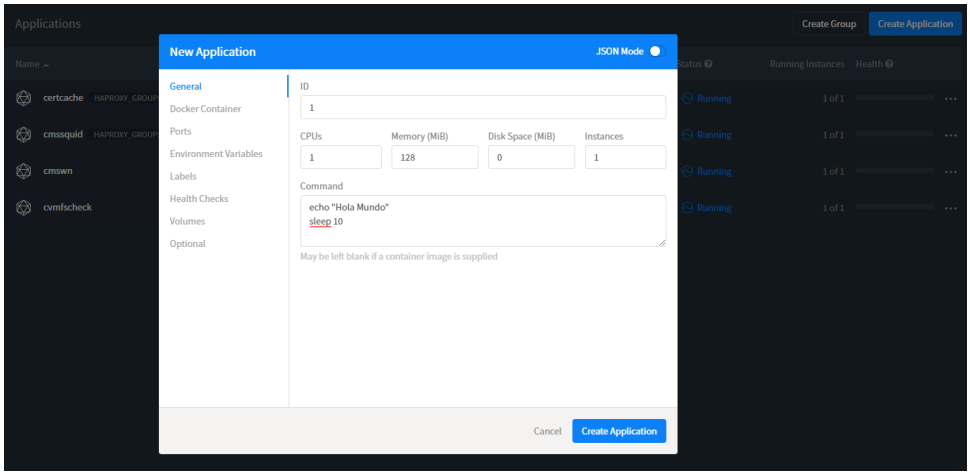

Figura 3.12: Creación de una aplicación en *Marathon*.

Figura 3.13: *Status* de la tarea en *Mesos*.

Para comprobar el estado de cada componente del entorno a desplegar se puede utilizar la API de Heat. El estado de los recursos (`resources` en el *template* de HOT) se puede mostrar a través de la línea de comandos con el siguiente comando:

```
$ heat resource-list <nombre-stack>
```

Varios de los recursos no completaban su estado debido a diversos problemas que se explican a continuación:

- El módulo *Docker_container* de *Ansible* no se reconocía en la versión inicial. *Docker_container* es un módulo de *Ansible* para crear contenedores Docker de las distintas aplicaciones que se definan en los roles de *Ansible*. La versión de *Ansible* instalada en la imagen que se usa como arranque es 2.0.0.2 y según [26], el módulo se implementó desde la versión 2.1. Por tanto la solución es instalar una nueva versión de *Ansible* al desplegar el entorno.

- "*Could not get lock /var/lib/dpkg/lock - open (11: Resource temporarily unavailable)*": Esto es debido a que hay más de un proceso tratando de instalar una aplicación, seguramente, debido a que las dependencias del fichero no se han cumplido o especificado correctamente. En la nueva versión del *template* que veremos a continuación no ha vuelto a ocurrir.

La segunda vez que adapté el *template*, el componente correspondiente a la instalación de los servicios de CMS en el nodo máster (`cms_services_deployment`) se quedaba bloqueado en el estado *CREATE_IN_PROGRESS*.

Para focalizar de dónde venía el problema seguí los siguiente pasos:

1. Lo primero que hice fue crear un nuevo servidor en la misma subred donde se crean los nodos de Mesos que hiciera de *proxy* para poder acceder a cada una de las instancias, y así posteriormente añadir un nuevo par de claves RSA en el nuevo servidor para poder acceder por SSH a las mismas (`ssh_key_name`).

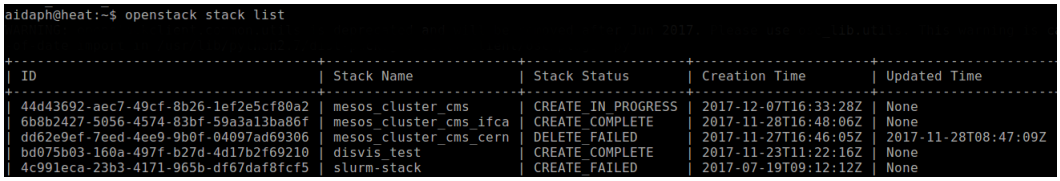

Figura 3.14: Estado de stack de Heat.

2. Una vez pude acceder a los servidores comprobé si el servicio *cloud-init* configuraba correctamente el arranque de cada uno de los servidores. Para ello, revisé el arranque con el comando **journalctl -a**.

3. Los servicios de arranque se iniciaban correctamente. Por tanto, lo siguiente que hice fue revisar el servicio *os-collect-config* que se conecta al servicio de metadatos y recoge toda la información para instalar todas las aplicaciones a través de *Ansible*. Revisé el mismo *log* que aparece con el comando *journalctl* y el servicio estaba correctamente arrancado pero vi que no llegaba a conectar con el servidor de metadatos.

4. Para revisar el servicio *os-collect-config* con más detalle se puede revisar el *log* del mismo como se muestra en *Listing* 3.10.

Listing 3.10: Log os-collect-config.

```
$ sudo service os-collect-config stop
$ sudo os-collect-config --force --one-time --debug
```

Revisando con detalle desde el *template* hasta los distintos *logs* del servidor con los desarrolladores de la solución dimos con el problema. La solución estaba adaptada para una versión de OpenStack anterior a la instalada en nuestra infraestructura local explicada en el Anexo A.2 y el problema correspondía a la línea 478 dentro de `cms_services_deploy` del *script*. En la versión anterior del *template* la línea aparecía como se muestra en *Listing* 3.11.

Listing 3.11: Versión anterior Server cms_services_deploy para DODAS.

```
server: { "Fn::Select" : [ '0',  get_attr:
[mesos_masters, attributes, mesos_server_id] ] }
```

El problema aparecía porque no podía asignarle un servidor para realizar la instalación y configuración de los servicios CMS. Modificando esta línea a la actual conseguí completar el despliegue del entorno.

Capítulo 4

Conclusiones y trabajo futuro

En este capítulo se explican cómo se han cumplido cada uno de los objetivos propuestos en el primer capítulo y cómo se ha alcanzado el próposito principal de este trabajo. Además, se detalla el alcance que puede tener este proyecto indicando algunos trabajos futuros que se van a desarrollar dentro de la línea del proyecto para poner en producción la solución en la infraestructura *cloud* del IFCA.

4.1. Conclusiones

El objetivo principal que se perseguía era crear un servicio *cloud* donde los usuarios de CMS pudieran ejecutar sus análisis. Para cumplir dicho objetivo he integrado y desplegado la solución DODAS desarrollada dentro del proyecto INDIGO-DataCloud en la infraestructura *cloud* del IFCA. Para poder integrar la solución al sistema local, primero, he tenido que adaptar el *template* de TOSCA inicial modificando los parámetros de cómputo, red y autenticación, así como varios parámetros de configuración para CMS. Una vez que he definido el fichero de TOSCA ya he creado una topología dónde se describen, uno a uno, los componentes del servicio, por tanto, si en algún futuro queremos modificar parte de la infraestructura, solo es necesario añadir o reconfigurar este *template*.

Una vez definida toda la aplicación que quiero ejecutar, se seleccionó un orquestador que se encargase del despliegue de todo el servicio. Debido a que la infrastructura *cloud* del IFCA se gestiona a través del proyecto OpenStack, se eligió como orquestador el servicio Heat para realizar esta función. Heat se ha elegido como servicio con el objetivo de que gestione una infraestructura para finalmente, poder dar un servicio a los usuarios, en este caso de CMS. Heat, en este momento, solo entiende el lenguaje HOT para poder desplegar toda la infraestructura por lo que he tenido que traducir el template inicial de TOSCA a HOT utilizando la herramienta *Heat-translator*. Antes de eso, he necesitado ponerme en contexto y entender cuál era el objetivo de TOSCA y cómo es la nomenclatura del lenguaje que utiliza, para así, después, ir modificando y añadiendo determinados parámetros de la infraestructura local. Una vez creado el fichero en HOT ya solo quedaba utilizar la API de Heat para crear el entorno. Debido a que surgieron diferentes problemas, tuve que adaptar parte del *template* de HOT, con ayuda de los desarrolladores de la solución de DODAS para conseguir que finalmente, Heat, desplegase el servicio. Una vez realizado este paso y comprobando que el entorno llegaba a completarse solo quedaba validar el servicio. Para ello, accedí a las web de Mesos y Marathon de mi servicio y comprobé que las aplicaciones de CMS estaban

ejecutándose a través de contenedores Docker. También lancé mi propia aplicación y vi que se ejecutaba en uno de los servidores que actuan como agentes en Mesos.

Por tanto, he conseguido desplegar un servicio para los usuarios de CMS del IFCA donde solo tengan que realizar un paso para ejecutar sus simulaciones y sus análisis a través de la web de Marathon. Todo el clúster que corre sus simulaciones se ejecuta por debajo de manera transparente a los usuarios. Además, otro de los objetivos era crear un clúster escalable y lo más eficiente posible. Para ello, en los *templates* que se le pasan como entrada a Heat se pueden definir el número de nodos que componen el clúster de Mesos a medida que se vaya incrementando la carga de aplicaciones en el servicio.

Para realizar todo este proyecto he tenido que conocer y aprender muchos de los conceptos que se usan en las tecnologías *cloud*. Una de las partes más complicadas ha sido entender la arquitectura de la solución, desde el paso inicial de creación de la topología hasta las diferentes aplicaciones que componen el clúster. Para ello, he necesitado conocer nuevos lenguajes como TOSCA y HOT así como el formato y la nomenclatura que se usa en cada uno de ellos y las diferencias entre ambos, sobre todo según el objetivo que se perseguía. Además de los lenguajes, he aprendido, y comprendido la arquitectura de varios servicios *cloud* dentro del proyecto OpenStack como Heat, Glance, Neutron o Nova.

Otro de los puntos, más importantes en la solución DODAS es el paso de autenticación a la hora de verificar la identidad de los usuarios y los servicios que se están implantando, por lo que he necesitado conocer el procedimiento que utilizan protocolos como *OpenID Connect* integrado dentro del sistema de Autenticación de INDIGO. Por último, he conocido la existencia y la arquitectura de nuevas herramientas como Mesos o Marathon, así como el uso y las ventajas que pueden ofrecer cada una de ellas en cuanto a escalabilidad y disponibilidad.

Actualmente, todo este trabajo ha formado parte de una fase de pruebas para comprobar que la solución desarrollada funciona correctamente y sea posible su puesta en producción en un futuro. Por consiguiente, este trabajo fin de máster ha contribuído dentro del proyecto INDIGO-DataCloud para integrar el servicio en una infraestructura real y que pueda llevarse a cabo y utilizarse próximamente.

4.2. Trabajo Futuro

Este trabajo fin de máster se ha realizado para evaluar la puesta en producción de un servicio que sea eficiente y con una facilidad de uso mayor que el que se está ofreciendo actualmente a través de la infraestructura *Grid*. En vista de ello y de que las conclusiones obtenidas son positivas, se van a tener que realizar varios pasos, algunos bastante complejos, para poder movernos hacia esta dirección. Ya se ha iniciado una primera puesta en marcha donde se han ido citando y comentando todos estos pasos con los desarrolladores de la solución, INFN, dentro del proyecto de INDIGO-DataCloud. La idea es que este servicio pueda ser funcional en el IFCA y pueda ser integrado en la Global Pool de HTCondor, no como *test site* sino como *local site* "real". Para ello, los desarrolladores y expertos en CMS y DODAS, se encargarán de testear toda la solución.

Otro de los pasos a realizar es la integración de una herramienta de monitorización como *Elastic Search* [27] dentro de la infraestructura *cloud* del IFCA para controlar localmente la carga del sistema y proporcionar la eficiencia y seguridad que el servicio

requiere.

Uno de los pasos más interesantes dentro de la solución sería eliminar el paso de traducción de TOSCA a HOT. *Heat* se basaría directamente en TOSCA para desplegar el entorno. La idea es la creación de un nuevo componente dentro del *pipeline* de WSGI [28] que haga la traducción automáticamente cuando le llega un *template* de TOSCA.

Por último, todo este trabajo futuro formará parte del proyecto EOSC-hub integrado en el marco H2020 dentro del proyecto europeo EOSC bajo el servicio ***EOSC-Hub DODAS thematic service***. EOSC-Hub tiene como principal objetivo la creación de un un sistema (*Hub*) dentro del proyecto *European Open Science Cloud*(EOSC) para que los usuarios de las comunidades científicas puedan descubrir, usar y acceder a los diferentes recursos que proporciona dicho proyecto. EOSC-hub crea un catálogo con más de 70 servicios donde contribuyen e-Infraestructuras nacionales, europeas y de todo el mundo.

En este proyecto, se conoce como *thematic service* a aquellos servicios a los que los usuarios pueden tener acceso para dar soporte a sus actividades científicas. Por ello, DODAS como *thematic service* contribuye a la comunidad de usuarios de CMS como servicio a la hora de que puedan acceder a los recursos de diferentes e-infraestructuras para desarrollar sus análisis.

Bibliografía

[1] Altamira Wiki IFCA https://grid.ifca.es/wiki/Altamira

[2] *LXPLUS Service* http://information-technology.web.cern.ch/services/lxplus-service

[3] *The NIST Definition of Cloud Computing.* Peter Mell, Timothy Grance. Special Publication 800-145, National Institute of Standars and Technology, September 2011.

[4] Álvaro López García. *"Scientific cloud Computing: Improved resource provisioning, intereoperability and federation"*, Universidad de Cantabria, Septiembre 2016.

[5] INDIGO-DataCloud. *"TOSCA template for specifying a deployment of a Mesos Cluster for CMS experiment"*, https://github.com/indigo-dc/tosca-templates/blob/master/mesos_cluster_cms.yaml

[6] Openstack Heat Translator https://wiki.openstack.org/wiki/Heat-Translator

[7] The Higgs Boson https://home.cern/topics/higgs-boson

[8] *Advancing Open Standards for the Information Society* (OASIS) https://www.oasis-open.org/

[9] OpenStack Foundation https://www.openstack.org

[10] YAML http://yaml.org

[11] Heat *Template Version* https://docs.openstack.org/heat/pike/template_guide/hot_spec.html#hot-spec-template-version

[12] Openstack Doc: *Template Guide for* Heat https://docs.openstack.org/heat/pike/template_guide/index.html

[13] Amazon Web Services: CloudFormation https://aws.amazon.com/es/cloudformation/

[14] Apache CloudStack https://cloudstack.apache.org/

[15] Rakesh Kumar, Kanishk Jain, Hitesh Maharwal, Neha Jain, Anjali Dadhich: *Apache CloudStack: Open Source Infrastructure as a Service Cloud Computing Platform.* In IJAETMAS, July 2014.

[16] *OpenStack Foundation*: Openstack Tacker. https://wiki.openstack.org/wiki/Tacker

[17] Docker. https://www.docker.com/

[18] Ansible https://www.ansible.com/

[19] Apache Mesos. http://mesos.apache.org/

[20] Apache Mesos: Software Projects built. http://mesos.apache.org/documentation/latest/frameworks/

[21] INDIGO-DataCloud. https://www.indigo-datacloud.eu/

[22] INDIGO IAM for indigo-dc. https://iam-test.indigo-datacloud.eu/

[23] *The INDIGO Token Translation Service*: WaTTs https://watts-dev.data.kit.edu/

[24] Twiki: *Software Guide on DODAS: Dynamic On Demand Analysis Service* https://twiki.cern.ch/twiki/bin/view/CMSPublic/DynamicOnDemandAS

[25] RFC 6749 - The OAuth 2.0 Authorization Framework https://tools.ietf.org/html/rfc6749

[26] Ansible: *Docker_container* http://docs.ansible.com/ansible/latest/docker_container_module.html#docker-container

[27] Elastic Search https://www.elastic.co/.

[28] WSGI: *Web Server Gateway Interface*. https://wsgi.readthedocs.io/en/latest/

[29] *Template* DODAS para la infraestructura IFCA *cloud* en TOSCA. https://raw.githubusercontent.com/aidaph/heat-cms/master/tosca/mesos_cluster_cms_ifca.yaml

[30] *OpendID Connect*. http://openid.net/connect/

[31] OpenStack Heat: *Software Deployment Image Script*. https://docs.openstack.org/heat/pike/template_guide/software_deployment.html#custom-image-script

[32] *Script* de configuración de Ansible para el despliegue de la aplicación CMS en el IFCA. https://raw.githubusercontent.com/indigo-dc/mesos-cluster/master/deploy/openstack-heat/setup.sh

[33] *Template* DODAS para la infraestructura IFCA *cloud* en HOT. https://raw.githubusercontent.com/aidaph/heat-cms/master/hot_cms_ifca.yaml

[34] OpenNebula https://opennebula.org/

[35] *Extensible Markup Language* (XML) https://www.w3.org/XML/

[36] PaaS Orchestrator https://www.indigo-datacloud.eu/paas-orchestrator

[37] Imágenes preconfiguradas de Ubuntu para Heat. http://cloud-images.ubuntu.com/releases/

Apéndice A

Template: Especificación de servicios en IFCA

La solución, así como las pruebas que se han ido explicando a lo largo de la memoria se han realizado a través de los servicios que se especifican a continuación.

A.1. Infraestructura IFCA-CSIC

La infraestructura local del IFCA donde actualmente se ejecutan los servicios CMS consiste en 5 racks con 266 servidores totales.
El primero de los racks consiste en 36 servidores Fujitsu PRIMERGY BX920 S2 con las siguientes características especificadas en el Cuadro A.1.

CPU	2 x Intel ®Xeon ®CPU X5670 @ 2.93GHz, 6 Cores per cpu
RAM	48 GB
Disco	250 GB
Red	4Gb Ethernet
Sistema Operativo	Ubuntu Xenial 16.04

Cuadro A.1: *CMS Rack1*

El segundo rack cuenta con 18 servidores Fujitsu PRIMERGY BX924 S4 especificado en el Cuadro A.2.

CPU	2 x Intel ®Xeon ®CPU E5-2697 v2 @ 2.70GHz, 12 cores per cpu
RAM	96 GB
Disco	400 GB
Red	2x10 Gb Ethernet
Sistema Operativo	Ubuntu Xenial 16.04

Cuadro A.2: *CMS Rack2*

El tercer *rack* consiste en 34 servidores HP ProLiant BL460c Gen8 con las siguientes características descritas en el Cuadro A.3.

El tercer *rack* consiste en 32 servidores SuperMicro MicroCloud 5037MC-H8TRF con las siguientes características descritas en el Cuadro A.4.

El último de los cinco *racks*, que se ha instalado recientemente, consiste en 144 servidores Intel Sandybridge E5-2670 con las siguientes características descritas en el

CPU	2 x Intel ®Xeon ®CPU E5-2697 v2 @ 2.70GHz, 12 cores per proc
RAM	128 GB
Disco	500 GB
Red	HP FlexFabric 10Gb 2-port 554FLB
Sistema Operativo	Ubuntu Xenial 16.04

Cuadro A.3: *CMS Rack3*

CPU	2 x Intel ®Xeon ®CPU E31260L @ 2.40GHz
RAM	16 GB
Disco	1 TB
Red	2 x 1 Gb Ethernet
Sistema Operativo	Ubuntu Xenial 16.04

Cuadro A.4: *CMS Rack4*

Cuadro A.5.

CPU	2xIntel Sandybridge E5-2670 2.6GHz
RAM	32 GB
Disco	500 GB
Red	1Gb Ethernet
Sistema Operativo	Ubuntu Xenial 16.04

Cuadro A.5: *Cloud*

A.2. Servicios Cloud IFCA-CSIC

En la infraestructura Cloud instalada en el IFCA se alojan los siguientes servicios necesarios para la implantación de la solución especificando, también las versión instalada. Todos estos servicios son desplegados a través de máquinas virtuales.

- ***Openstack*** *Application Programming Interface* (API) : 3.8.1
- *Openstack Identity Service (**Keystone**)*: Liberty 2.3.1
- *Openstack Volume Service (**Cinder**)*: Ocata 1.11.0
- *Openstack Image Service (**Glance**)*: Ocata 2.6.0
- *Openstack Orquestration Service (**Heat**)* 1.8.0 sobre Ubuntu Xenial 16.04.2.

Apéndice B

Template: Entorno CMS en TOSCA

Listing B.1: Definición del Entorno de Mesos de CMS a través de TOSCA.

```
tosca_definitions_version: tosca_simple_yaml_1_0

imports:
  - indigo_custom_types:
    https://raw.githubusercontent.com/indigo-dc/
    tosca-types/master/custom_types.yaml

description: TOSCA template for DODAS in
             IFCA cloud infraestructure.

topology_template:

  inputs:
    iam_token:
      type: string
      default: "my_token"

    iam_client_id:
      type: string
      default: "myclient_id"

    iam_client_secret:
      type: string
      default: "myclient_secret"

    cms_local_site:
      type: string
      default: "T2_ES_IFCA"

    cms_stageoutsite:
      type: string
```

```
      default: "T2_ES_IFCA"

    cms_stageoutserver:
      type: string
      default: "srmfe01.ifca.es"

    cms_stageoutprefix:
      type: string
      default: "srm://srmfe01.ifca.es:8444
      /srm/managerv2?SFN=/cmsdisk/"

    cms_stageoutsite_fallback:
      type: string
      default: "DUMMY"

    cms_stageoutserver_fallback:
      type: string
      default: "DUMMY"

    cms_stageoutprefix_fallback:
      type: string
      default: "DUMMY"

    cms_input_path:
      type: string
      default: "DUMMY"

    cms_input_protocol:
      type: string
      default: "xrootd"

    onedatatoken:
      type: string
      default: "DUMMYTOKEN"

    onedatacache:
      type: string
      default: "160.44.194.19"

    monitordb_ip:
      type: string
      default: "193.146.75.99:8080"

    elasticsearch_secret:
      type: string
      default: "5z0087gnSf"

    imageCMS:
```

```
      type: string
      description: Image used to boot a server
      default: ubuntu16-heat-ifca

    networkCMS:
      type: string
      description: Network ID for the servers
      default: 2b729f05-cda2-4316-bab6-4247aa06d708

  node_templates:

    mesos_master:
      type: tosca.nodes.indigo.MesosMaster
      properties:
        marathon_password: test_pass
        chronos_password: test_pass
        mesos_masters_list: { get_attribute:
         [ mesos-master-server, private_address ] }
      requirements:
        - host: mesos-master-server


    cms_services:
      type: tosca.nodes.indigo.CmsServices
      properties:
        marathon_password: test_pass
        mysquid_host: { get_attribute:
        [ mesos-lb-server, private_address, 0 ] }
        proxycache_host: { get_attribute:
        [ mesos-lb-server, private_address, 0 ] }
        iam_access_token: { get_input: iam_token }
        iam_client_id: { get_input: iam_client_id }
        iam_client_secret: { get_input:
        iam_client_secret }
        cms_local_site: { get_input: cms_local_site }
        onedatatoken: { get_input: onedatatoken }
        onedatacache: { get_input: onedatacache }
      requirements:
        - host: mesos_master


    mesos_slave:
      type: tosca.nodes.indigo.MesosSlave
      properties:
        master_ips: { get_attribute:
        [ mesos-master-server, private_address ] }
        front_end_ip: { get_attribute:
        [ mesos-master-server, private_address, 0 ] }
```

```
    requirements:
      - host: mesos-slave-server


  cms_wn:
    type: tosca.nodes.indigo.CmsWnConfig
    properties:
      mysquid_host: { get_attribute:
      [ mesos-lb-server, private_address, 0 ] }
      proxycache_host: { get_attribute:
      [ mesos-lb-server, private_address, 0 ] }
      cms_local_site: { get_input:
      cms_local_site }
      cms_stageoutsite: { get_input:
      cms_stageoutsite }
      cms_stageoutserver: { get_input:
      cms_stageoutserver }
      cms_stageoutprefix: { get_input:
      cms_stageoutprefix }
      cms_stageoutsite_fallback: { get_input:
      cms_stageoutsite_fallback }
      cms_stageoutserver_fallback: { get_input:
      cms_stageoutserver_fallback }
      cms_stageoutprefix_fallback: { get_input:
      cms_stageoutprefix_fallback }
      cms_input_protocol: { get_input:
      cms_input_protocol }
      cms_input_path: { get_input:
      cms_input_path }
      monitordb_ip: { get_input: monitordb_ip }
      elasticsearch_secret: { get_input:
      elasticsearch_secret }
    requirements:
      - host: mesos_slave


  mesos_load_balancer:
    type: tosca.nodes.indigo.MesosLoadBalancer
    properties:
      master_ips: { get_attribute:
      [ mesos-master-server, private_address ] }
      marathon_password: test_pass
    requirements:
      - host: mesos-lb-server

  mesos-master-server:
    type: tosca.nodes.indigo.Compute
    capabilities:
```

```
      endpoint:
        properties:
          network_name: PUBLIC
          dns_name: mesosserverpublic
          ports:
            mesos_port:
              protocol: tcp
              source: 5050
            marathon_port:
              protocol: tcp
              source: 8080
      scalable:
        properties:
          count: 1
      host:
        properties:
          num_cpus: 4
          mem_size: 8 GB
      os:
        properties:
          image: { get_input: imageCMS }
          #networks:
          #- network: { get_param: network }


  mesos-slave-server:
    type: tosca.nodes.indigo.Compute
    capabilities:
      scalable:
        properties:
          count: 3
      host:
        properties:
          num_cpus: 2
          mem_size: 4 GB
      os:
        properties:
          image: { get_input: imageCMS }
          #networks:
          #- network: { get_param: network }


  mesos-lb-server:
    type: tosca.nodes.indigo.Compute
    capabilities:
      endpoint:
        properties:
          network_name: PUBLIC
```

```
            dns_name: mesoslb
        scalable:
          properties:
            count: 1
        host:
          properties:
            num_cpus: 2
            mem_size: 2 GB
        os:
          properties:
            image: { get_input: imageCMS }
            #networks:
            #- network: { get_param: network }

outputs:
  mesos_lb_ip:
    value: { get_attribute: [ mesos-lb-server,
    public_address ] }
  mesos_endpoint:
    value: { concat: [ 'http://', get_attribute:
    [ mesos-master-server, public_address, 0 ],
    ':5050' ] }
  marathon_endpoint:
    value: { concat: [ 'http://', get_attribute:
    [ mesos-master-server, public_address, 0 ],
    ':8080' ] }
```

Glossary

AAI Authentication and Authorization Infrastracture. 32

API Application Programming Interface. 36, 39, 42, 45

CMS Compact Muon Selenoid. 1, 2, 5, 19, 21, 22, 28, 29, 31, 32, 38, 41, 43–46, 49, 50

DODAS Dynamic on Demand Analysis Service. 21, 22, 26, 27, 46

EOSC European Open Science Cloud. 1

ESFRI European Strategy Forum on Research Infrastructures. 1

HOT Heat Orchestration Template. 10–12, 16, 17, 19, 32, 36, 37, 40–42, 45, 46

IaaS Infraestructure as a Service. 2, 6–8, 11

IAM Identity and Access Management. 31–33, 35

IFCA Instituto de Física de Cantabria. 1, 2, 6, 12, 19, 28, 45, 46, 49

INDIGO-DataCloud INtegrating Distributed data Infrastructures for Global ExplOtation. 6, 19, 21, 22, 28, 29, 31–35, 38, 45, 46

INFN Istituto Nazionale di Fisica Nucleare. 6, 29, 46

PaaS Platform as a Service. 2, 6, 7, 21

SaaS Software as a Service. 2, 6, 7

TOSCA Topology and Orchestration Specification for Cloud Applications. 14–17, 19, 21, 22, 28, 32, 33, 36, 37, 42, 45, 46

TTS Token Translation Service. 22, 35

WSGI Web Server Gateway Interface. 46