ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Grado

Despliegue automático de servicios de datos climáticos con Kubernetes
(Automatic deployment of climate data services with Kubernetes)

Para acceder al Título de

Graduado en Ingeniería de Tecnologías de Telecomunicación

Autor: Pablo Celaya Crespo

Septiembre - 2019

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

Realizado por: Pablo Celaya Crespo
Director del TFG: Antonio S. Cofiño González
Título: "Despliegue automático de servicios de datos climáticos con Kubernetes"
Title: "Automatic deployment of climatic data services with Kubernetes"

Presentado a examen el día: 30 de septiembre de 2019

para acceder al Título de

GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre): Luis Muñoz Gutiérrez
Secretario (Apellidos, Nombre): Alberto Eloy García Gutiérrez
Vocal (Apellidos, Nombre): Antonio S. Cofiño González

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFG (sólo si es distinto del Secretario)
Vº Bº del Subdirector	Trabajo Fin de Grado Nº (a asignar por Secretaría)

Agradecimientos

Al Grupo de Meteorología y Computación de Santander, y en particular al director de este trabajo Antonio Cofiño, por darme la oportunidad de colaborar con ellos todo este tiempo y aprender tanto durante el proceso.

A todos los miembros del CEDA, especialmente Philip Kershaw y Matt Pryor, por permitirme trabajar con ellos y ayudarme en todo lo necesario.

A todos mis profesores del Grado estos últimos tres años, por su trabajo y esfuerzo que me han dado los conocimientos para desarrollar las competencias que me han llevado hasta aquí.

Muchas gracias a todos.

Resumen

En este trabajo se describe un caso de uso de herramientas para la automatización del despliegue de servicios en un entorno *DevOps*[1]. En concreto se hablará de la utilización de *Kubernetes* como herramienta de orquestación de contenedores de *software* (i.e. *Docker*) para el despliegue de servicios y aplicaciones de *software* de acceso a datos climáticos.

Este trabajo ha sido llevado a cabo como parte de una estancia en el *Centre for Environmental Data Analysis* (CEDA), del Reino Unido. Las tareas han sido realizadas dentro de un proyecto para el *Copernicus Climate Change Service* (C3S), que forma parte del Programa Copernicus, dirigido conjuntamente por la Agencia Espacial Europea (ESA) y la Agencia Europea de Medio Ambiente (EEA), en representación de la Unión Europea. El servicio C3S promueve el acceso, libre y abierto, a datos medioambientales para que puedan ser usados por la comunidad científica, proveedores y desarrolladores de servicios.

El objetivo de este trabajo es describir un modelo que facilita el acceso a los datos, y probar la eficiencia de *Kubernetes* como herramienta para la automatización del despliegue y el autoescalado horizontal de servicios, que se ejecutan en el *back-end*.

[1] Acrónimo inglés de *development* (desarrollo) y *operations* (operaciones). Metodología de ingeniería de *software* cuyo objetivo es unificar el trabajo de los desarrolladores y los profesionales de sistemas de tecnologías de la información. Defiende la automatización y el monitoreo en todas las etapas de la construcción del *software*: desarrollo, pruebas, despliegue y administración.

Abstract

This document describes a use case of tools for the automation of service deployment in a *DevOps*[2] environment. In particular, the use of *Kubernetes* will be discussed as a software container orchestration tool for the deployment of climate data access services and software applications.

The tasks in this document have been carried out as part of an industrial placement in the Centre for Environmental Data Analysis (CEDA), in United Kingdom. The jobs accomplished here belong to a project of the Copernicus Climate Change Service (C3S), which is part of the Copernicus Program, leaded by the European Space Agency (ESA) and the European Environment Agency (EEA) on behalf of the European Union. C3S supports access, free and open, to environmental data to be used by the scientific community, providers and service suppliers.

The goal of this document is describing a model that provides access to data and proving the efficiency of *Kubernetes* as a tool for the automatic deployment and the horizontal autoscaling of services running in the back-end.

[2] Acronym for Development and Operations. Software engineering methodology which aims to unify the work of software developers and information–technology operations professionals. It defends automation and monitoring on all software development stages: development, testing, deployment and management.

Índice general

Capítulo 1: Introducción ... 3
1.1 Motivación ... 3
1.2 Estructura del trabajo ... 3
Capítulo 2: Conceptos previos ... 4
2.1 Contenedores software ... 4
2.2 Kubernetes ... 6
2.2.1 Objetos de Kubernetes ... 7
2.2.2 Arquitectura ... 10
2.2.3 Horizontal Pod Autoscaler ... 12
2.2.4 Helm ... 13
2.3 Arquitectura ESGF ... 15
2.4 Birdhouse ... 17
2.4.1 Web Processing Services ... 17
Capítulo 3: Infraestructura y servicios ... 20
3.1 JASMIN ... 20
3.1.1 Servicios Cloud en JASMIN ... 20
Capítulo 4: El Programa Copernicus ... 23
4.1 Copernicus Climate Change Service (C3S) ... 24
4.1.1 Climate Data Store ... 25
4.1.2 Climate Projections for the Climate Data Store ... 27
Capítulo 5: Implementación Técnica ... 29
5.1 Configuración del autoescalado de servidores THREDDS para un nodo ESGF ... 31
5.2 Despliegue de Birdhouse mediante un *Chart de Helm* ... 34
5.3 Demostración del autoescalado con *Kubernetes* ... 36
5.4 Conclusiones ... 41
5.5 Líneas futuras ... 41
Bibliografía ... 43

Índice de figuras

Figura 1 - Evolución de la instalación de aplicaciones 4
Figura 2 - Ejemplo de manifest de un Deployment en Kubernetes 8
Figura 3 - Ejemplo de manifest de un Service en Kubernetes 9
Figura 4 - Arquitectura de Kubernetes 10
Figura 5 - Ejemplo de plantilla para una Chart de Helm 14
Figura 6 - Ejemplo de fichero `values.yaml` con los valores de configuración por defecto 14
Figura 7 - ESGF: Arquitectura distribuida 15
Figura 8 - Servicios y componentes de un nodo ESGF 16
Figura 9 - Interacción con WPS para caso síncrono 18
Figura 10 - Componentes de Birdhouse 19
Figura 11 - Infraestructura JASMIN 20
Figura 12 - Arquitectura de la nube de JASMIN 21
Figura 13 - Infraestructura del Climate Data Store 26
Figura 14 - CP4CDS: Acceso y procesado de datos para el CDS 27
Figura 15 - CP4CDS: Interfaces para el CDS 28
Figura 16 - Infraestructura federada para CP4CDS 28
Figura 17 - Fichero `cluster.yaml` con los parámetros de configuración para RKE 31
Figura 18 - Template `hpa.yml` para el autoescalado horizontal de servidores THREDDS . 33
Figura 19 - Pull Request creada en el repositorio oficial del proyecto “ESGF Docker” 34
Figura 20 - Fichero `custom.cfg` con los parámetros de configuración de Phoenix 35
Figura 21 - Uso de memoria de los Pods THREDDS 38
Figura 22 - Uso de CPU de los Pods THREDDS 38
Figura 23 - Peticiones por segundo y número de réplicas durante la prueba 39
Figura 24 - Uso de memoria RAM de los Pods Phoenix 40
Figura 25 - Uso de CPU de los Pods Phoenix 40
Figura 26 - Peticiones por segundo y número de réplicas 40

Capítulo 1: Introducción

1.1 Motivación

En los últimos años en la industria de desarrollo de software, se ha experimentado un creciente grado de aceptación en el uso de contenedores *software*, debido a que ofrecen un modelo de implementación basado en imágenes que se adapta perfectamente a la cadena de desarrollo y despliegue de aplicaciones. Junto a esta idea de aislar las aplicaciones en contenedores, han surgido soluciones para administrar grupos de contenedores que ofrecen un servicio en conjunto, lo cual se conoce como orquestación. La herramienta más usada para ello es Kubernetes, el motor de orquestación de contenedores ideado por Google.
El trabajo recogido en este documento se ha llevado a cabo como parte de una estancia en un centro de investigación del Reino Unido donde, entre otras labores, se desarrolla *software* para el acceso y procesado de datos climáticos y que en los últimos años ha investigado y desarrollado el uso de estas tecnologías. Las tareas realizadas en este Trabajo de Fin de Grado (TFG), recogidas bajo un proyecto a nivel europeo dentro del Programa Copernicus, se centran en implementar nuevos modelos de despliegue que otorguen nuevas capacidades a los sistemas en producción y realizar pruebas con el fin de demostrar con métricas la eficiencia de dichas soluciones.

1.2 Estructura del trabajo

Este documento se ha dividido en cinco capítulos. Tras este breve capítulo introductorio, en el capítulo 2 se dan a conocer los conceptos previos indispensables para la comprensión del trabajo realizado; se explica qué es un contenedor *software*, se detalla la arquitectura de un clúster *Kubernetes* así como algunos de sus componentes y otros términos específicos de la tecnología y por último se trata de resumir en qué consisten las soluciones *software* desplegadas: el nodo ESGF y *Birdhouse.*
En el capítulo 3, se comenta cómo es y cómo funciona la infraestructura de computación empleada, conocida como JASMIN.
El capítulo 4 trata de poner en contexto el trabajo realizado informando sobre la iniciativa europea Copernicus. Se introducirá su proyecto CP4CDS, que ha motivado las labores realizadas, así como el *Climate Data Store*, que centra los esfuerzos del proyecto.
Por último, en el capítulo 5 se habla de la parte práctica del trabajo. La implementación técnica, resumida en tres tareas, explica el desarrollo de nuevos modelos de despliegue para aplicaciones que prestan servicios de acceso y procesado de datos climáticos. En primer lugar, se explica la puesta a punto del entorno de desarrollo desplegando un clúster *Kubernetes* en la infraestructura utilizada. A continuación, se comenta el trabajo desarrollado, el *software* utilizado y las aportaciones al proyecto generadas por este trabajo. Finalmente, se presenta una demostración de las soluciones planteadas tras ejecutar realizar pruebas sobre ellas y recoger los resultados en forma de gráficas.

Capítulo 2: Conceptos previos

2.1 Contenedores software

La virtualización, en referencia al acto de simular recursos computacionales, ha sido utilizada desde sus inicios en la industria del desarrollo de *software*.
Originalmente, las aplicaciones se desplegaban directamente sobre la máquina física, que contaba con un único sistema operativo, por lo que surgían problemas a la hora de adaptar el *software* a los múltiples sistemas y al gestionar las dependencias con librerías, etc.
Más adelante llegaron las máquinas virtuales, que permitían tener varias instancias de una aplicación ejecutándose sobre la misma máquina física, sobre diferentes sistemas operativos y que proporcionaban cierto nivel de aislamiento. Sin embargo, al ser una máquina virtual una emulación completa de un sistema físico, hardware y sistema operativo completo (kernel, librerías, etc.), su utilización conlleva una sobrecarga importante e implica la necesidad de tener un hipervisor (más sobrecarga) gestionando las máquinas virtuales para que compartan los recursos *hardware* del sistema físico.
Los contenedores *software* son un tipo de virtualización a nivel de sistema operativo en el que se comparte el kernel con el anfitrión, por lo que se reduce enormemente la sobrecarga que aparece con las máquinas virtuales. Un contenedor consiste en una unidad de *software* que contiene todos los componentes necesarios para correr una aplicación, su código y dependencias. Su principal cualidad es la portabilidad, ya que son muy ligeros (en comparación con las máquinas virtuales) y permiten garantizar un entorno en el que una aplicación se ejecutará correctamente. Por el contrario, en aras de reducir la sobrecarga proporcionan un nivel de abstracción mucho menor que las máquinas virtuales, por lo que son inherentemente más inseguros.
En la Figura 1 a continuación se incluye una comparación de la evolución en el despliegue de aplicaciones en función del modelo utilizado.

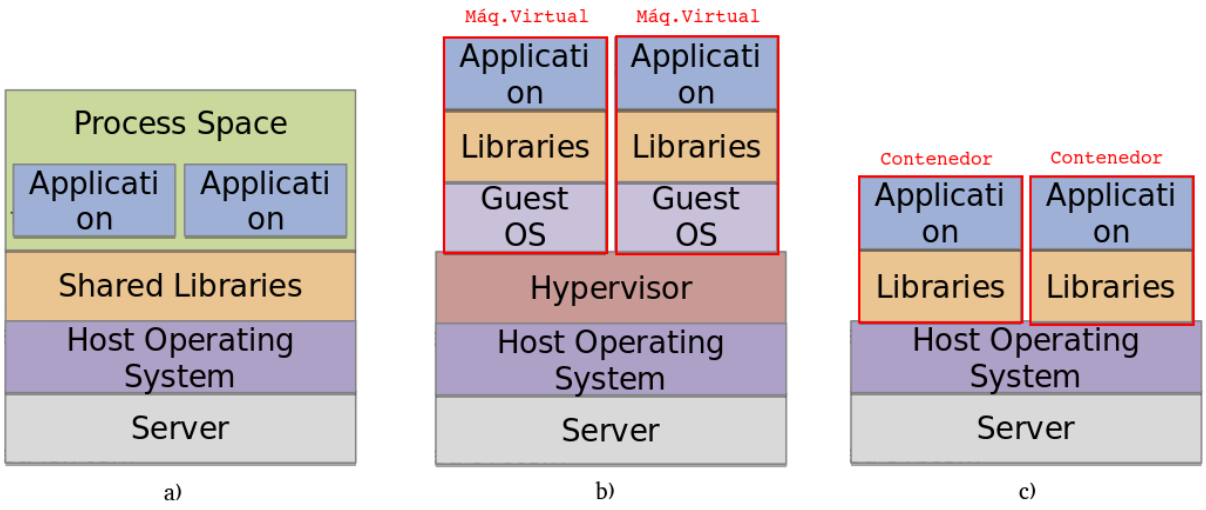

Figura 1 - Evolución de la instalación de aplicaciones. a) Instalación tradicional. b) Uso de máquinas virtuales. c) Uso de contenedores (fuente: [1])

Aunque estrictamente no sea una definición correcta, se puede entender en cierta manera un contenedor como una versión ligera de una máquina virtual, y de hecho así suele aparecer

en la literatura de referencia. Para ver las auténticas diferencias entre ambas se presenta la Tabla 1, resumiendo algunas de las propiedades más comunes que se incluyen habitualmente en estudios comparativos de ambos modelos [2].

Parámetro	Máquinas Virtuales	Contenedores
Sistema operativo	Se ejecutan sobre hardware virtualizado y cargan un kernel en su propia región de memoria	Todos los contenedores comparten el kernel con el anfitrión
Comunicación	Se comunican a través de dispositivos Ethernet virtualizados	Se pueden implementar mecanismos de comunicación entre procesos como señales, tuberías, sockets, etc.
Rendimiento	Sufren una sobrecarga considerable debido a la necesidad de traducir las instrucciones máquina	Proporcionan un rendimiento prácticamente igual al del sistema operativo anfitrión
Aislamiento	La compartición de librerías, ficheros, etc. está muy restringida o no es posible	Se pueden montar directorios de forma transparente
Tiempo de inicio	Iniciar una máquina virtual lleva un tiempo del orden de minutos	Los contenedores pueden arrancar en segundos en comparación
Almacenamiento	Precisan de mucho espacio para todo el sistema operativo además de los programas a instalar y ejecutar	Al compartir el sistema operativo, precisan de mucho menor espacio
Seguridad	Depende de la implementación del hipervisor	Pueden requerir de mecanismos adicionales de seguridad para no comprometer al anfitrión

Tabla 1 - Comparación entre máquinas virtuales y contenedores software

Existen múltiples tecnologías de contenedores: *Docker, Linux Containers* (LXC)*, singularity*, etc. siendo *Docker* la más utilizada. Desde un punto de vista técnico, *Docker* (al igual que el resto) utiliza dos prestaciones del kernel de Linux para la implementación de lo que finalmente se conoce como contenedor *software*:

- *namespaces*: característica del kernel que proporciona aislamiento a diferentes niveles entre los procesos. Esto permite que cada contenedor tenga sus propios identificadores de proceso (PID), su propia dirección IP privada o su propio espacio de usuarios.
- *control groups*: conocido como *cgroups,* proporciona control sobre la compartición de recursos como CPU, memoria, etc. y se utiliza como mecanismo para gestionar dichos recursos, limitando su uso.

Docker utiliza otras prestaciones como, por ejemplo, el filtrado de llamadas al sistema (con módulos del kernel como *apparmor o seccomp*) para aportar un nivel extra de seguridad.
Los contenedores se distribuyen en imágenes, que son instantáneas consistentes en una serie de capas definidas por un desarrollador para construir una aplicación, desde el sistema

operativo base hasta el código final pasando por dependencias, librerías, etc. Así, un contenedor se puede definir también como una instancia de una imagen ejecutándose.
Docker también proporciona la capacidad de buscar, descargar y compartir imágenes fácilmente; el lugar donde se almacenan las imágenes se conoce como registro. *Docker* ofrece un registro público (el *Docker Registry*[3]) donde los desarrolladores pueden subir sus imágenes con *software* listo para ejecutarse como servidores web, bases de datos, entornos de desarrollo, etc. También es posible desplegar un registro de imágenes privado propio para controlar quién puede descargar o subir imágenes.

En definitiva, como se puede apreciar un contenedor *software* otorga muchas facilidades a la hora de distribuir y desplegar todo tipo de *software*, incluido el que se trata en este TFG y por ello es importante comprender en qué consiste y cómo se utiliza.

2.2 Kubernetes

Kubernetes [3] es un *software* de código abierto, desarrollado originalmente por *Google*, para la orquestación y el despliegue de aplicaciones *contenerizadas*. *Kubernetes* proporciona el *software* necesario para construir y desplegar sistemas fiables, escalables y distribuidos.

Ya han sido mencionadas las ventajas de utilizar contenedores para ejecutar y agrupar aplicaciones software. Sin embargo, en un entorno de producción, es importante la gestión de los contenedores en ejecución para minimizar el tiempo de indisponibilidad de un servicio, por ejemplo. Es aquí donde *Kubernetes* entra en acción para, si un contenedor falla, iniciar uno nuevo automáticamente. A continuación, se listan algunas de las capacidades que *Kubernetes* puede proporcionar:

- Descubrimiento de servicios y equilibrio de carga: *Kubernetes* puede exponer un contenedor usando un nombre de dominio propio o utilizando su propia dirección *IP*. También es capaz de equilibrar la carga de trabajo y distribuir el tráfico de manera que el despliegue sea estable.
- Orquestación de almacenamiento: *Kubernetes* permite montar automáticamente multitud de sistemas de almacenamiento; almacenamiento local, proveedores *cloud* públicos, etc.
- Despliegues y restablecimientos automáticos: se puede automatizar *Kubernetes* para crear contenedores y eliminar los ya existentes asignando sus recursos a los nuevos contenedores. Esta función es realmente útil cuando hay que realizar una actualización en el software; además puede configurarse de tal manera que el servicio siempre esté disponible durante el proceso.
- Reparación automática: *Kubernetes* reinicia los contenedores que fallan, los reemplaza y elimina aquellos contenedores que no responden a las comprobaciones definidas por el usuario, denominadas *health checks*.
- Gestión de la información de configuraciones: *Kubernetes* permite almacenar y gestionar la información relativa a la configuración de los contenedores, así como la información más sensible como contraseñas, claves, *tokens,* etc. Esta información sensible y parámetros de configuración pueden ser actualizados sin necesidad de

[3] https://hub.docker.com/

reconstruir las imágenes de los contenedores, y sin necesidad de exponer en abierto la información sensible.

2.2.1 Objetos de Kubernetes

Los objetos de *Kubernetes* [4] son entidades utilizadas para representar el estado del clúster. En concreto, definen qué aplicaciones están corriendo (y en qué nodos[4]), los recursos disponibles para esas aplicaciones y las políticas respecto a cómo dichas aplicaciones han de comportarse (políticas de acceso, de reinicio, actualizaciones o tolerancia ante fallos).
Una de las características de *Kubernetes* es que sus objetos son definidos y configurados de forma declarativa, no de forma imperativa. Es decir, el desarrollador no define una serie de tareas a realizar para desplegar un servicio, sino que declara el estado deseado para el despliegue. Es responsabilidad de *Kubernetes* el asegurarse de que ese estado se cumpla en todo momento.
Todos los objetos en *Kubernetes* constan de metadatos (datos que identifican unívocamente al objeto como un nombre, espacio de nombres asociado, etiquetas[5]…), una especificación (conocido en *Kubernetes* como *spec*, que es el estado deseado para el objeto o las características que se quiere para él) y un estado (conocido como *status*, que describe el auténtico estado del objeto). Para crear un objeto, se ha de interactuar con la API de *Kubernetes* proporcionando metadatos (al menos, un nombre) y el *spec* en formato JSON dentro del campo de la petición. Lo más habitual es utilizar un cliente de línea de comandos, *kubectl*, al que se le pasa un fichero en formato *YAML* (llamado *manifest*) que posteriormente se convierte en JSON para hacer la petición a la API. Sin embargo, existen multitud de clientes para diferentes entornos y lenguajes de programación.
En la Figura 2 se puede observar un ejemplo de estos ficheros YAML con la descripción de un objeto de tipo *Deployment*, en el que se especifica que se requieren dos réplicas de un servidor web *Nginx* ejecutándose dentro de sendos contenedores. Como campos requeridos aparecen siempre `apiVersion` (con la versión de la API de Kubernetes que se pretende utilizar), `kind` (con el tipo de objeto descrito), `metadata` y `spec`. La especificación es diferente para cada tipo de objeto, y está definida en la documentación de referencia de la API[6] de acuerdo a la versión utilizada (siendo la versión 1.16 la más reciente a fecha de la redacción de este TFG).

[4] En Kubernetes, un nodo es una máquina física o virtual donde finalmente se ejecutarán los contenedores. Un conjunto de nodos forma un clúster Kubernetes.
[5] Las etiquetas de Kubernetes son parejas clave-valor que pueden asociarse a un objeto para identificarlo y agruparlo con otros objetos.
[6] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
labels:
  app: nginx
  env: prod
spec:
  selector:
    matchLabels:
      app: nginx
      env: prod
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
        env: prod
    spec:
      containers:
      - name: nginx
        image: nginx:1.17.3
        ports:
        - containerPort: 80
```

Figura 2 - Ejemplo de manifest de un Deployment en Kubernetes

A continuación, se listan los principales objetos de Kubernetes, necesarios para comprender su funcionamiento básico, así como el trabajo realizado:

- *Pods*
  Es la unidad básica de *Kubernetes*, el objeto más pequeño y más simple desplegable en su modelo. Un *Pod* encapsula un contenedor software (o más de uno), recursos de almacenamiento y recursos de red (dirección IP única y puertos TCP/UDP). El *Pod* representa una instancia individual de una aplicación dentro de *Kubernetes* que puede consistir en un solo contenedor (lo más común y recomendable) o en un pequeño número de contenedores estrechamente relacionados y que comparten recursos.
- *Controllers*
  Estos objetos crean y gestionan múltiples *Pods*, manejando la réplica y el lanzamiento, y proporcionando capacidad de reparación automática. Por ejemplo, si un nodo falla, el *Controller* puede automáticamente reemplazar un *Pod* planificado en dicho nodo con un recambio idéntico en un nodo diferente. Existen distintos tipos de *Controllers,* como por ejemplo los *Deployments* (los más utilizados), *StatefulSets* (para aplicaciones con estado, que guardan datos relativos a sus sesiones) o *DaemonSets* (se aseguran que todos los nodos tienen una réplica del *Pod*, útiles para *Pods* de monitorización o *logging*, por ejemplo).
- *Service*
  Es una forma abstracta de exponer una aplicación ejecutándose en una serie de *Pods* como un servicio en red. Con los *Services*, *Kubernetes* asigna a un conjunto de *Pods* su propia dirección IP y nombre de dominio y puede equilibrar la carga entre ellos. La motivación para la existencia de los *Services* está en que los *Pods* en *Kubernetes*

tienen un ciclo de vida limitado. Son creados y destruidos dinámicamente (por ejemplo, por un *Deployment*) y esto puede crear un problema si un conjunto de *Pods* (por ejemplo, un *front-end*) depende de otro conjunto de *Pods* (un *back-end*) dentro del clúster. El conjunto de *Pods* objetivo de un *Service* se determina mediante un selector, que es un filtro basado en etiquetas de *Kubernetes* definido por el usuario.
Existen distintos tipos de *Services:*

- *ClusterIP:* expone el servicio en una dirección IP virtual interna al clúster *Kubernetes* que solamente es accesible desde los objetos de este.
- *NodePort*: expone el servicio en una dirección IP física y un mismo puerto TCP/UDP en todos los nodos del clúster.
- *LoadBalancer:* este servicio fue creado para funcionar con proveedores de servicios en la nube pública o privada. Expone el servicio utilizando un equilibrador de carga externo, que puede crearse automáticamente.

En la Figura 3 se muestra un ejemplo de un *manifest* para un *Service* de *Kubernetes*:

```
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 38080
      targetPort: 80
```

Figura 3 - Ejemplo de manifest de un Service en Kubernetes

El controlador de este *Service* escanea continuamente *Pods* que coincidan con el selector (cualquier *Pod* con la etiqueta `app=nginx`) y dirige el tráfico en el puerto `38080/TCP` en todos los nodos del clúster hacia dichos *Pods* en el puerto `80/TCP`.

➢ *Ingress*

Un *Ingress* expone rutas HTTP y HTTPS desde fuera del clúster hacia los *Services* de *Kubernetes* y puede proporcionar URLs accesibles desde el exterior, equilibrado de carga, terminación de TLS y *hosting* virtual basado en nombres. Un *Ingress* es la forma más utilizada para exponer externamente servicios HTTP (puerto 80/TCP) o HTTPS (puerto 443/TCP). Para otros puertos y/o protocolos, por lo general, se utilizan los *Services* de tipo *NodePort* o *LoadBalancer.* Para que un *Ingress* pueda cumplir con su cometido necesita de un *Ingress Controller* desplegado en el clúster.

➢ *Ingress Controller*

Consiste en un controlador que implementa la funcionalidad de los *Ingress*, generalmente mediante un equilibrador de carga. Así como el *Ingress* es una abstracción para los servicios, el *Ingress Controller* no es simplemente un objeto

abstracto de *Kubernetes* sino que despliega un contenedor dentro de un *Pod* en el clúster. Existen múltiples fabricantes de tecnología de equilibrado de carga que han desarrollado su propio *Ingress Controller* (*HAProxy*, *Nginx*, *F5*…) [5].

- *Volumes*

  *L*os ficheros en disco dentro de un contenedor son efímeros, debido a la propia naturaleza de los contenedores, lo que puede suponer un problema para ciertas aplicaciones. Cuando un contenedor falla, *Kubernetes* lo reinicia con un estado limpio, por lo que los ficheros se pierden; además, cuando dentro de un *Pod* corren múltiples contenedores, es frecuente que necesiten compartir ficheros. El objeto *Volume* resuelve ambas problemáticas. En esencia, un *Volume* es simplemente un directorio con algo de información en su interior que es accesible a los contenedores dentro de un *Pod*. Cómo se crea ese directorio, el medio que lo respalda y su contenido están determinados por el tipo particular de *Volume* que se emplee. Para usar este objeto, se ha de especificar en el *spec* de un *Pod* qué *Volumes* se proporcionan (campo *.spec.volumes* del *manifest*) y dónde y cómo montarlo en el contenedor (campo *.spec.container.volumeMounts*).

  Existen distintos tipos de objetos *Volume*: algunos de ellos trabajan con los sistemas de almacenamiento de los proveedores de nube pública (tipos *azureDisk, awsElasticBlockStore, gcePersistentDisk…)* o privada (*cinder)*, otros montan ficheros o directorios del sistema de ficheros del nodo (*hostPath)* y otros utilizan sistemas de ficheros distribuidos (*nfs*, *cephfs)*. Existen dos tipos adicionales que son muy utilizados en *Kubernetes*: *configMap* para inyectar parámetros de configuración a los *Pods* y el tipo *secret* para pasar información sensible como contraseñas y claves privadas.

2.2.2 Arquitectura

Los elementos que conforman la arquitectura de un clúster *Kubernetes* (ver Figura 4) se dividen en dos categorías: componentes maestros y componentes de nodo.

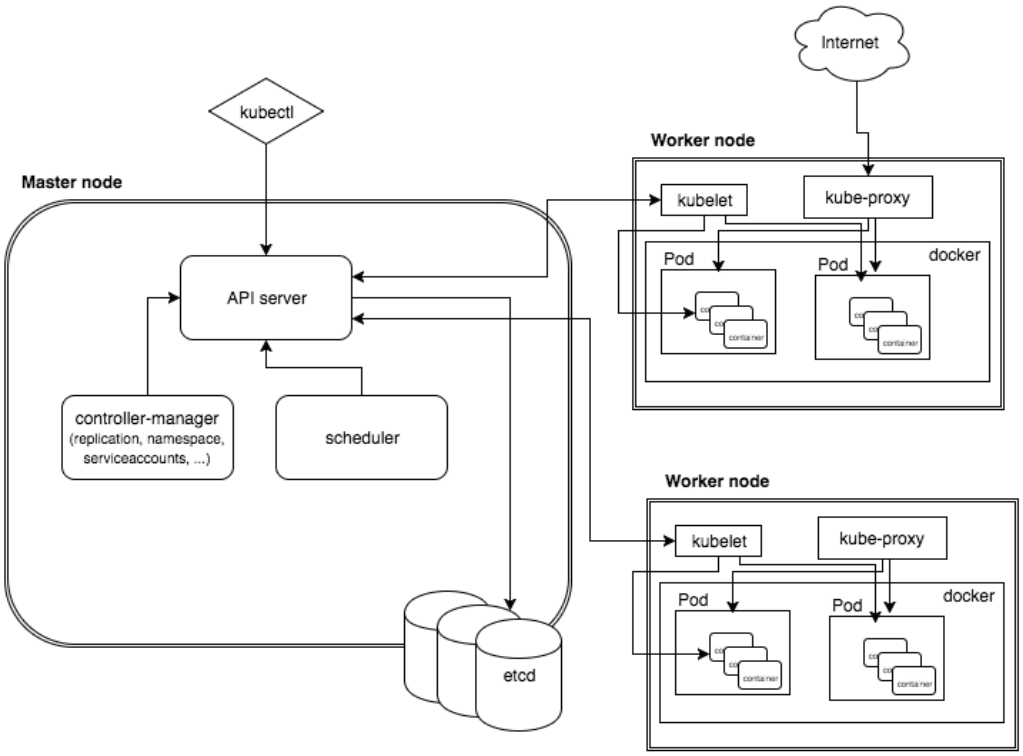

Figura 4 - Arquitectura de Kubernetes (fuente: [6])

Componentes maestros

Estos componentes son los que toman las decisiones globales relativas al clúster y detectan y responden ante diferentes eventos. Estos componentes pueden ejecutarse en cualquier máquina del *clúster*; sin embargo, por simplicidad, a la hora de desplegar un clúster *Kubernetes* se suelen iniciar siempre en la misma máquina, y no se suelen ejecutar contenedores de usuario en dicha máquina.

- *kube-apiserver*: este componente expone la API de *Kubernetes*, actuando como el front-end del plano de control.
- *etcd*: almacenamiento consistente y de alta disponibilidad basado en parejas clave valor que se utiliza como almacenamiento de respaldo para todos los datos del clúster.
- *kube-scheduler*: monitoriza los *Pods* creados que no tienen asignados un nodo y selecciona un nodo para que puedan ejecutarse. Tiene en cuenta diversos factores para la toma de decisiones, que incluyen desde requerimientos de recursos hardware, localidad de los datos, especificaciones de afinidad (y de no afinidad) por parte del usuario, etc.
- *kube-controller-manager*: consiste en un único proceso que engloba varios controladores:
  - Controlador de nodos - responsable de notificar y responder cuando un nodo deja de funcionar.
  - Controlador de réplicas - responsable de mantener el número correcto de *Pods* en el sistema.
  - Controlador de *endpoints* - une *Pods* y servicios de *Kubernetes*.
  - Controlador de cuentas de servicios y *tokens* - crea cuentas por defecto y *tokens* de acceso a la API para cada espacio de nombres.

Componentes de nodo

Estos componentes se ejecutan en todos los nodos en los que corren los contenedores del usuario, permitiendo el mantenimiento de los *Pods* y proporcionando un entorno de contenedores software a *Kubernetes*.

- *kubelet*: agente que se ejecuta en cada nodo del clúster. Se asegura de que los contenedores están corriendo dentro de un *Pod*.
- *kube-proxy*: *proxy* de red que implementa parte del concepto servicio de *Kubernetes*. Mantiene una serie de reglas de red que permiten la comunicación de los *Pods* desde sesiones dentro o fuera del clúster. En principio utiliza la capa de filtrado de paquetes disponible en el sistema operativo si está disponible, en caso contrario redirige el tráfico él mismo.
- *Container runtime:* software responsable de la ejecución los contenedores. *Kubernetes* soporta diferente software para este propósito: *Docker*, *containerd*, *cri-o*, *rktlet*…

Además de los componentes mencionados, un clúster de *Kubernetes* cuenta con lo que se conoce como *addons*, que utilizan recursos de *Kubernetes* (*DaemonSet, Deployment, Pod,* etc.) para implementar prestaciones para el clúster. El principal es el *clúster DNS*, un servidor DNS, que coexiste con otros DNS configurados en el entorno, para servir registros DNS a los servicios de *Kubernetes*. Los contenedores iniciados por *Kubernetes* automáticamente incluyen este servidor en sus búsquedas (por ejemplo, en el fichero */etc/resolv.conf*).

Otros ejemplos de *addons* incluyen una interfaz web con datos sobre el clúster (conocida como *Dashboard*), monitorización de los recursos que consumen los contenedores o un servicio de *logs* a nivel de clúster.

2.2.3 Horizontal Pod Autoscaler

Un componente no mencionado hasta ahora y que merece especial atención por su gran relevancia en las tareas desempeñadas como parte de este TFG, es el *Horizontal*[7] *Pod Autoscaler* (HPA) [7]. Este componente sirve para escalar de forma automática el número de *Pods* bajo un *Controller* (por ejemplo, las réplicas en un *Deployment*) en función de la(s) métrica(s) que se le proporcione. Atendiendo a la documentación de referencia de la API en relación a este componente[8], se aprecia que en su versión estable solamente soporta como métrica el uso de CPU, sin embargo también aparecen dos versiones en fase Beta[9] que soportan otras métricas conocidas por *Kubernetes* de forma nativa (memoria RAM) o proporcionadas por terceros.
Está implementado como un ciclo de control que ajusta el número de réplicas para que la utilización media de CPU (o la métrica correspondiente) permanezca por debajo de un límite establecido por el usuario, operando sobre la ratio entre el valor de métrica deseado y el valor observado, con una tolerancia configurable por el usuario (10% por defecto):

$$replicasDeseadas = \left\lceil replicasObservadas \times \frac{valorMetricaObservada}{valorMetricaDeseada} \right\rceil$$

El HPA tiene en cuenta una serie de parámetros para su funcionamiento, configurables en el *kube-controller-manager*, que es el componente que consulta la utilización de recursos de los *Pods* que aparecen en la declaración del HPA. Los valores que aparecen a continuación son los valores por defecto:

- ➔ Un período de 15 segundos entre cada consulta del valor de una métrica
- ➔ Un tiempo de espera de 30 segundos desde que el *Pod* aparece como disponible antes de intentar el autoescalado, para esperar a que el *Pod* sea estable.
- ➔ Un tiempo de espera de 5 minutos desde la última vez que se rebajó el número de réplicas hasta que se realiza la siguiente rebaja. Este parámetro se utiliza para controlar lo que se conoce como *thrashing*, que es un fenómeno por el cual el número de réplicas variaría continuamente debido a una rápida fluctuación de los valores de las métricas, provocando inestabilidad en el sistema.

Este componente goza de gran importancia en este TFG debido a que uno de los objetivos marcados es probar la eficiencia de *Kubernetes* en el escalado de servicios, lo cual se lleva a cabo empleando el *Horizontal Pod Autoscaler*.

[7] Cuando se habla de escalado horizontal de servicios, se hace referencia a la creación de nuevas instancias del servicio. El escalado vertical, por otro lado, consiste en asignar más recursos de CPU y/o memoria al mismo número de instancias.
[8] https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.15/#horizontalpodautoscaler-v1-autoscaling
[9] En *Kubernetes*, un recurso de la API en versión Beta es un objeto que ha sido probado y verificado, cuyo soporte se va a seguir manteniendo pero cuya estructura o sintaxis pueden estar sujetos a cambios en el futuro.

2.2.4 Helm

Helm [8] es un gestor de paquetes para *Kubernetes*. Permite instalar, configurar y actualizar aplicaciones de *Kubernetes* de gran complejidad con múltiples componentes, facilitando también su creación y publicación. La última versión de *Helm* es mantenida por la CNCF[10] en colaboración con *Microsoft*, *Google*, *VMWare* y su propia comunidad de desarrolladores.
Helm utiliza un formato en el empaquetamiento de aplicaciones denominado *Charts* (que se traduce como "tablas"). Las *Charts* son una colección de ficheros con una estructura definida que describen una serie de recursos de *Kubernetes* a ser desplegados. Una única *Chart* puede ser utilizada para desplegar algo tan simple como un *Pod* con un servidor FTP o la pila completa de una aplicación web al uso (servidores HTTP, bases de datos, cachés, etc.). Una *Chart* debe crearse dentro de un directorio con el nombre de la misma y responder a la siguiente estructura:

```
<nombre>/
  Chart.yaml     # Fichero YAML con información descriptiva de la Chart
  LICENSE        # Opcional: fichero de texto con la licencia de uso
  README.md      # Opcional: fichero tipo README con instrucciones de uso
  requirements.yaml    # Opcional: Fichero YAML que lista las posibles
                       # dependencias con otras Charts
  values.yaml    # Fichero YAML con valores de configuración por defecto
  charts/        # Directorio que contiene otras Charts de las que depende
  templates/     # Directorio con las plantillas que, combinadas con los
                 # valores de values.yaml, genera manifests de Kubernetes
```

El fichero `values.yaml` y las plantillas son la piedra angular de las *Charts* de *Helm*. Estas se escriben utilizando un paquete del lenguaje de programación *Go*[11], que permite asignar valores a los campos de los *manifests* de forma dinámica, así como la utilización de funciones para entre otras cuestiones dar formato, realizar filtrados, insertar bloques de control, etc.
A continuación se incluye la descripción del *Deployment* del ejemplo del apartado *2.2.1*, pero en este caso en forma de plantilla de una *Chart* (Figura 5) y cómo podría ser el fichero `values.yaml` correspondiente (Figura 6):

[10] *Cloud Native Computing Foundation*, un proyecto de la *Linux Foundation* para promocionar el uso de contenedores. Se fundó en el año 2015 con el lanzamiento de *Kubernetes* 1.0.
[11] https://golang.org/pkg/text/template/

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
labels:
  app: nginx
  env: {{ .Values.nginx.labels.env }}
spec:
  selector:
    matchLabels:
      app: nginx
      env: prod
  replicas: {{ .Values.nginx.replicas }}
  template:
    metadata:
      labels:
        app: nginx
        env: {{ .Values.nginx.labels.env }}
    spec:
      containers:
      - name: nginx
        image: {{ .Values.nginx.image}}:{{ .Values.nginx.version}}
        ports:
        - containerPort: {{ .Values.nginx.port }}
```

Figura 5 - Ejemplo de plantilla para una Chart de Helm

```
---
nginx:
  labels:
    env: prod
  replicas: 4
  port: 80
```

Figura 6 - Ejemplo de fichero `values.yaml` con los valores de configuración por defecto

Helm consta de dos componentes: un cliente (`helm`) y un servidor (`tiller`). El cliente se ejecuta en línea de comandos para el desarrollo y evaluación de *Charts*, la gestión de repositorios de *Charts* y la interacción con el servidor para el despliegue. El servidor corre normalmente en el clúster *Kubernetes* e interacciona con el cliente sirviendo como interfaz con la API de *Kubernetes*. En definitiva, el cliente es responsable en la gestión de *Charts* y el servidor es responsable en la gestión de lo que se conoce como *releases*, que son instancias de *Charts* en ejecución.
Al ejecutar una *Chart* para su despliegue, se le debe pasar al cliente el directorio raíz de la misma (o bien la referencia del repositorio donde esté alojada) y opcionalmente valores que sobreescriban los valores por defecto.

Este TFG trabaja sobre *Charts* de *Helm* para definir el despliegue y escalado de los servicios, de ahí la importancia de describir su funcionamiento y estructura.

2.3 Arquitectura ESGF

Esta arquitectura de software tiene gran importancia en este trabajo, ya que uno de los objetivos propuestos es probar la eficiencia de *Kubernetes* para el autoescalado horizontal de servidores THREDDS, que como se verá a continuación constituyen una parte importante de lo que se conocen como “Nodos ESGF”.

ESGF [9], *Earth System Grid Federation*, son las siglas que representan la colaboración de distintas agencias, grupos e instituciones alrededor del mundo dedicados al desarrollo y operación de un sistema a largo plazo para la gestión, acceso y análisis de datos climáticos. El *ESGF* administra una base de datos descentralizada para el manejo de datos científicos con múltiples petabytes distribuidos en docenas de lugares alrededor del mundo.

La arquitectura de ESGF está basada en el paradigma peer-to-peer (*P2P*) (ver Figura 7), a través de una serie de emplazamientos (llamados “Nodos”) que están distribuidos geográficamente pero que pueden operar entre sí debido a que adoptan una serie de servicios, protocolos e interfaces en común. Los nodos intercambian información sobre los datos que albergan y los servicios que ofrecen, y confían entre ellos para el registro de usuarios y el establecimiento del control de acceso. Los datos y metadatos son gestionados y almacenados de manera independiente en cada nodo, pero el usuario accede a ellos como si se tratara de un único archivo global.

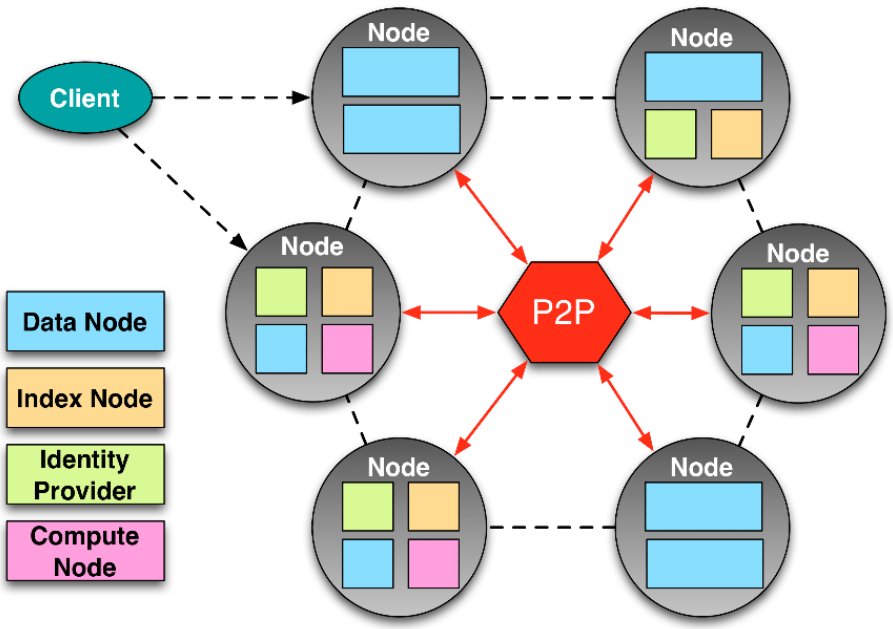

Figura 7 - ESGF: Arquitectura distribuida (fuente: [2])

Internamente cada nodo ESGF está compuesto de una serie de servicios y aplicaciones que colectivamente permiten el acceso a los datos y la gestión de los usuarios. Los distintos componentes de software están agrupados en cuatro áreas de funcionalidad que se instalan de manera modular:

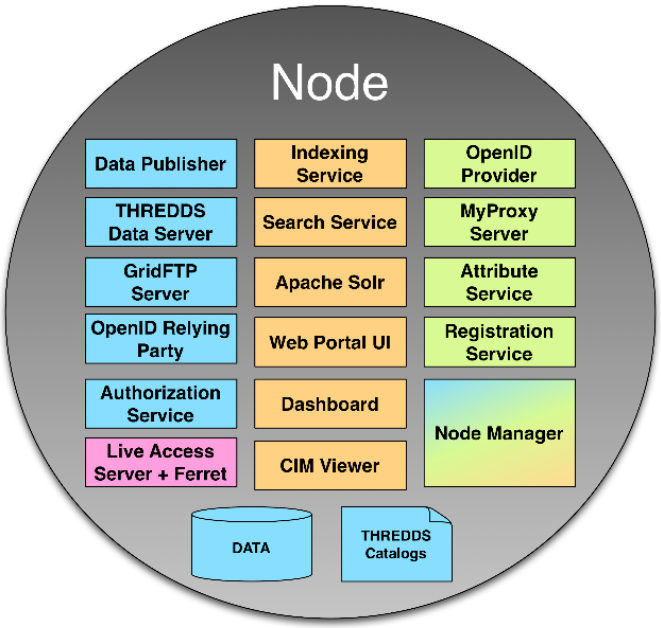

Figura 8 - Servicios y componentes de un nodo ESGF (fuente: [10])

- **Nodo de datos**: incluye los servicios para garantizar la publicación y el acceso a los datos. De color azul en la Figura 8.
  - *Data publisher*: genera los catálogos de metadatos. Escanea los datos almacenados en el nodo y los vuelve disponibles a través del sistema.
  - *THREDDS Data Server*: permite el acceso a los datos y metadatos del ESGF. Desarrollado por Unidata.
  - *GridFTP Server*: sirve datos utilizando un protocolo basado en FTP para permitir una transferencia de alto rendimiento, confiable y autenticada de forma segura. Desarrollado por Globus.
  - *OpenID Relying Party and Authorization Service*: asegura una correcta autenticación y autorización.
- **Nodo de indexado**: contiene los servicios para el indexado y búsqueda de metadatos. De color naranja en la Figura 8.
  - *Indexing service*: servicio de indexado; analiza los metadatos disponibles en un repositorio y los guarda en el *back-end*.
  - Search service: servicio de búsqueda; consulta los metadatos indexados y devuelve los resultados con información descriptiva así como los puntos de acceso disponible.
  - *Apache Solr*[12]: aplicación web que actúa como motor de búsqueda.
  - *Web Portal UI*: interfaz de usuario a través del navegador web.
  - *Dashboard*: sistema distribuido de monitorización para ESGF. Es responsable de recolectar información histórica sobre el estado de los nodos.
- **Proveedor de identidad**: permite la autenticación y securiza la entrega de atributos a los usuarios. De color verde en la Figura 8.

[12] https://lucene.apache.org/solr/

- ➢ *OpenID Provider*: registra a los usuarios y los autentica en el sistema, incluyendo funcionalidad *Single-Sign-On*[13] para accesos a través del navegador.
- ➢ *MyProxy Server*: es utilizado para crear certificados temporales que pueden ser utilizados por los usuarios en librerías y herramientas para autenticarse durante la petición de datos.
- ➢ *Attribute Service* y *Registration Service*: servicio para garantizar que el usuario cuenta con los permisos necesarios para garantizar el cumplimiento de las políticas de control de acceso.

❖ **Nodo de computación**: posiblemente el elemento más importante de la arquitectura, contiene servicios de alto nivel para el análisis y visualización de los datos. De color rosa en la Figura 8.

2.4 Birdhouse

Birdhouse [11] es un proyecto colaborativo de software abierto. Consiste en un *framework* que contiene una colección de servicios de procesado de datos específicos para la comunidad científica del clima. Cobra importancia en este proyecto porque otro de los objetivos propuestos es el diseño de una *Helm Chart* (ver apartado *2.2.4*) para el despliegue de algunos de los servicios que incluye *Birdhouse*. Estos servicios siguen el estándar de la OpenGis Consortium (OGC), denominado *Web Processing Services* (WPS). El análisis de datos por parte de la comunidad científica del clima, como ya se ha mencionado, requiere por lo general del procesado de grandes volúmenes de datos. Una manera común de proceder por parte de los científicos es descargar los datos de algún proveedor (por ejemplo, un *Data Node* de ESGF) y ejecutar los análisis en su propia infraestructura, “en local”. Sin embargo, con el continuo aumento de los datos disponibles, este proceso cada vez es menos conveniente y surgen soluciones para llevar a cabo los procesos de análisis allí donde se encuentran los datos.

2.4.1 Web Processing Services

Web Processing Services (WPS) [12] es un estándar de la *Open Geospatial Consortium* (OGC), organización sin ánimo de lucro dedicada a diseñar estándares abiertos de calidad para la comunidad geoespacial, que define cómo implementar un modelado o cálculo geográfico como un servicio web.

La interfaz WPS estandariza entre otros conceptos la manera de definir los procesos y sus datos de entrada y salida, cómo un cliente puede solicitar la ejecución de un proceso y cómo han de entregarse las salidas de éste. WPS utiliza tres operaciones básicas, basadas en los métodos *GET* y *POST* de HTTP (ver Figura 9):

1. La interacción inicial utiliza la operación *GetCapabilities* para devolver la lista de procesos ofrecida por el *Web Processing Service*.
2. Los parámetros de entrada requeridos por el proceso y los resultados de salida se definen a través de la operación *DescribeProcess*.
3. La operación *Execute* permite al usuario ejecutar el proceso.

[13] Procedimiento de autenticación que habilita a un usuario determinado para acceder a varios sistemas con una sola instancia de identificación.

Existen también *WPS* asíncronos para los cuales se definen las operaciones *GetStatus* y *GetResult* para obtener el estado de un proceso en ejecución y sus resultados una vez finalizado, respectivamente.

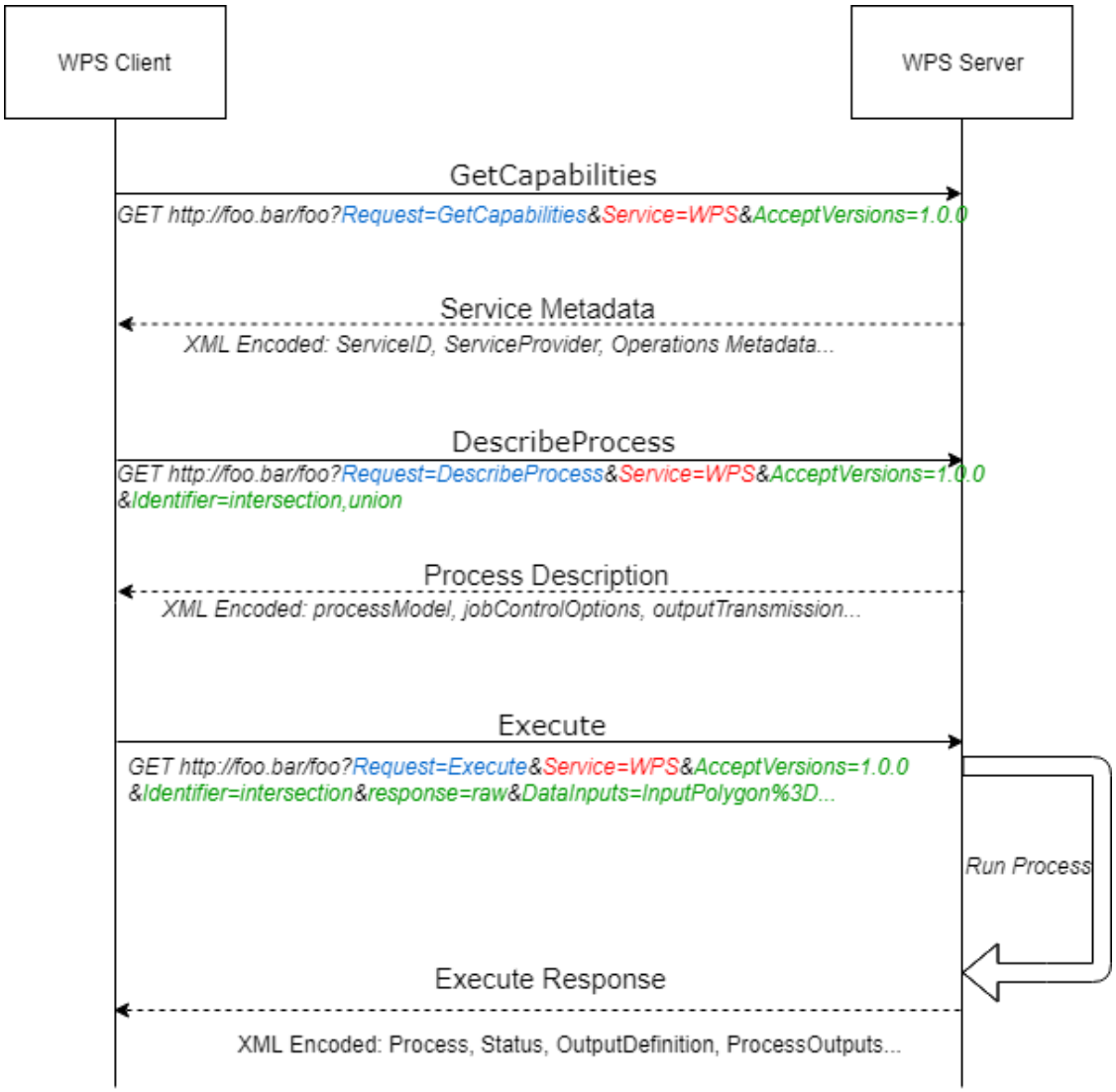

Figura 9 - Interacción con WPS para caso síncrono

El principal objetivo de *Birdhouse* [13] es facilitar el análisis de datos como servicio. Para ello proporciona una serie de herramientas para poder disponer de la infraestructura necesaria para el despliegue de *Web Processing Services*. Los distintos componentes de *Birdhouse* se organizan en módulos independientes llamados "*pájaros" (birds),* que por lo general se centran en problemáticas o temas concretos. Por ejemplo, se han desarrollado *birds* para proyectos centrados en la evaluación de eventos relacionados con climas extremos, como *Black Swan* [14]. *Birdhouse* utiliza la implementación en lenguaje Python de WPS, PyWPS 4.0[14], y ha sido desarrollado por el *Deutsche Klimarechenzentrum (DKRZ)* dentro de un proyecto para el C3S, donde *Birdhouse* se utiliza como *back-end* en el procesado de datos procedentes de las proyecciones climáticas.

Algunos de los componentes de *Birdhouse* son (ver Figura 10):

[14] https://pywps.org/2016/12/07/pywps-4.0.0.html

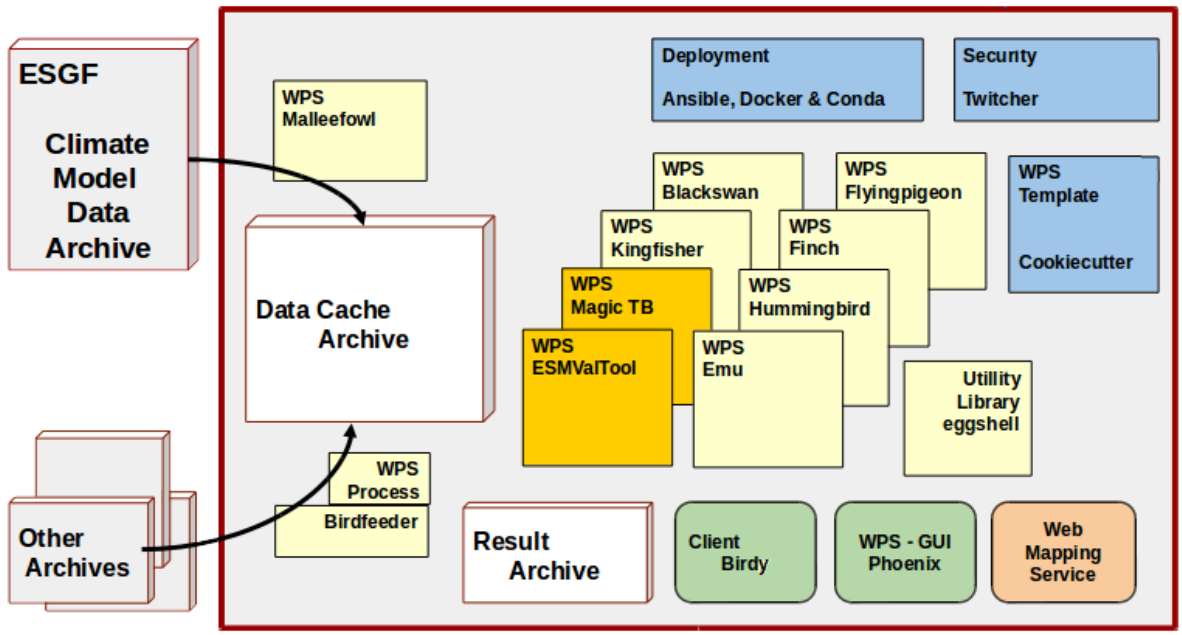

Figura 10 - Componentes de Birdhouse (fuente: [11])

Componentes de la parte cliente:

- *Phoenix*: cliente WPS basado en Web que dispone de una interfaz gráfica para el usuario y acceso a datos del *ESGF*.
- *Birdy*: cliente de línea de comandos para WPS.

Componentes de la parte servidor:

- *Emu*: procesos WPS simples que sirven de ejemplo para demostraciones.
- *Black Swan*: servicios para evaluación de eventos relacionados con clima extremo.
- *Kingfisher*: servicios para el análisis de datos de observación de la Tierra.
- *Finch*: servicios para el cálculo de índices climáticos.

Multitud de proyectos de investigación utilizan el *framework Birdhouse*, tal y como se detalla en [13], entre ellos el ya mencionado *Climate Projections for the Climate Data Store* (*CP4CDS*), que centra los esfuerzos de este trabajo.

Capítulo 3: Infraestructura y servicios

3.1 JASMIN

Como ya se ha mencionado, las tareas de este trabajo han sido llevadas a cabo en el Centro para el Análisis de Datos Ambientales (CEDA), en Reino Unido. Este grupo de investigación utiliza una infraestructura de computación conocida como JASMIN[15].
La infraestructura JASMIN (*Joint Analysis System Meeting Infrastructure Needs*) [15] [16] es un "super-data-clúster" creado para suplir los requisitos de análisis de datos de la comunidad de modelización de sistemas climáticos y terrestres del Reino Unido y Europa.
La infraestructura JASMIN proporciona computación y almacenamiento unidos entre sí por una red de gran ancho de banda. Utiliza una topología única con unas capacidades mucho mayores que las de un centro normal de datos. Esta infraestructura está formada por un sistema central (el súper clúster de datos) y tres sistemas satelitales en las universidades de Leeds, Bristol, y Reading. Cada sistema satelital consiste en discos de 100, 150 y 500 terabytes de almacenamiento, respectivamente, y en recursos de computación.

Como se puede apreciar en la Figura 11, los servicios de JASMIN se pueden dividir en dos clases diferentes. Por un lado, se encuentran los servicios de computación y análisis (color naranja) y, por otro lado, se encuentran los servicios de datos (color verde). Como se ha comentado previamente, ambos servicios son gestionados por el CEDA.

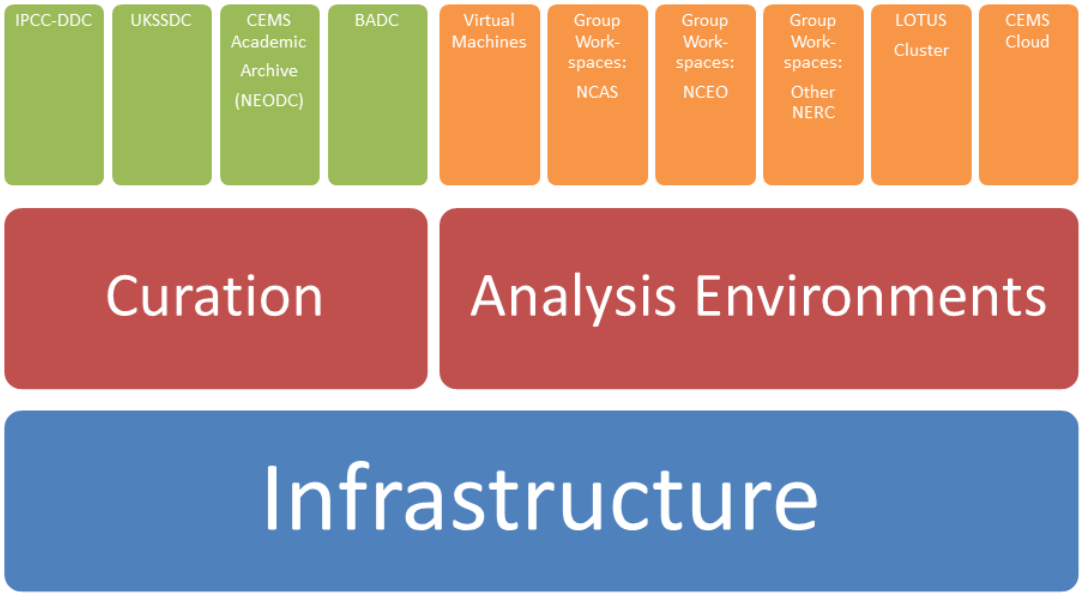

Figura 11 - Infraestructura JASMIN (fuente: [15])

3.1.1 Servicios Cloud en JASMIN

JASMIN también proporciona un servicio de computación basado en la nube construido sobre Openstack, un software de código abierto utilizado para la provisión de nubes privadas o híbridas. La nube de JASMIN [17] es similar al concepto general de nube en cuanto a que permite a instituciones y proyectos el consumo de recursos computacionales como servicio

[15] http://jasmin.ac.uk/

sin necesidad de aprovisionar y mantener la infraestructura física. Además, los usuarios pueden crear sus propias máquinas virtuales dentro de la infraestructura JASMIN.
La característica más importante de la nube de JASMIN es que tiene acceso al sistema de almacenamiento basado en Panasas. Por este motivo, es ideal para proyectos que trabajan con datos almacenados allí.
La Figura 12 representa los componentes que forman la estructura JASMIN.

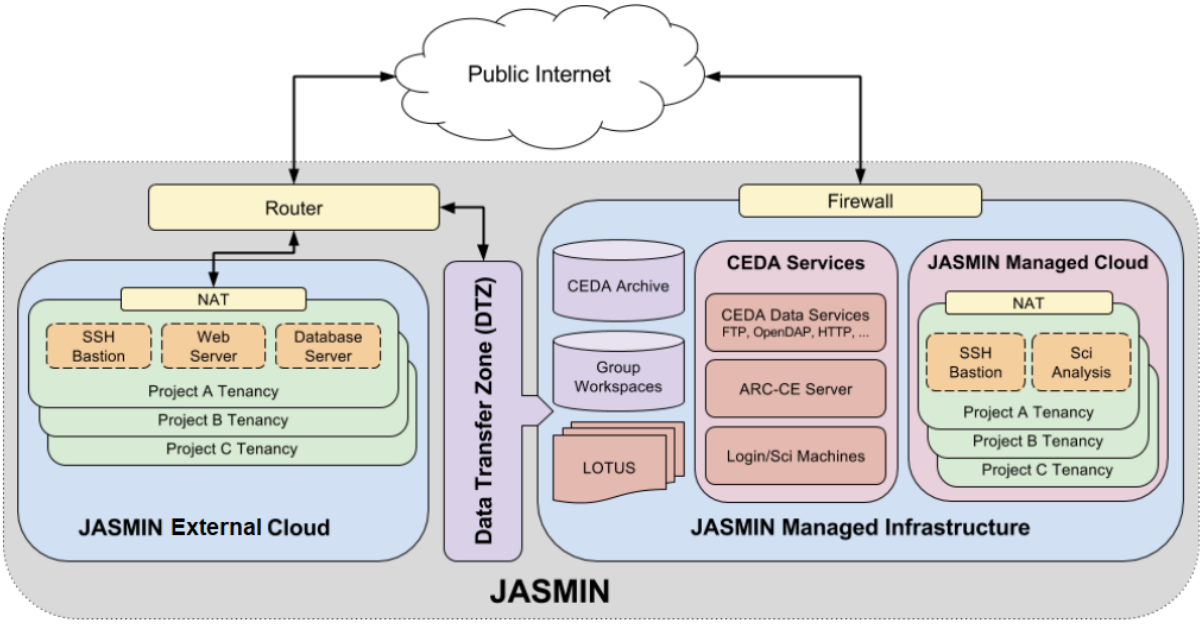

Figura 12 - Arquitectura de la nube de JASMIN (fuente: [17])

En la Figura 2 puede observarse que por un lado se encuentra la parte no administrada (*external*) y por otro, la administrada (*managed*).
La primera de ellas se ofrece como una *IaaS* (*Infrastructure as a Service*), y se encuentra fuera del firewall de JASMIN. Los usuarios tienen acceso como *root* a las máquinas, y recae en ellos toda la responsabilidad de la administración del sistema. Además, por encontrarse fuera del firewall, los usuarios no tienen acceso directo al sistema de almacenamiento de JASMIN. Por el contrario, estos usuarios sí tienen acceso a la zona de transferencia de datos (DTZ) que les conecta con los servicios integrados del CEDA.
La segunda parte que conforma la infraestructura (*managed*) se ofrece como una PaaS (*Platform as a Service)*. Se encuentra dentro del firewall de JASMIN, por lo que los usuarios tienen acceso directo al sistema de almacenamiento. Para prevenir que dichos usuarios realicen cambios en el sistema, no se les asignan privilegios de acceso *root* a sus máquinas y solo pueden desplegar máquinas virtuales con una plantilla predefinida. Sin embargo, estos no tienen que hacerse cargo de la seguridad de las mismas. Actualmente solo hay dos plantillas disponibles: *SSH bastion*, una máquina de inicio de sesión, y *Sci Analyisis,* un servidor de análisis científico con una configuración similar a los servicios compartidos de análisis científico de JASMIN, que cuenta con el archivo de datos del CEDA ya montado.
Ambas partes comparten una estructura de red similar. Cada *tenancy* (conjunto de recursos computacionales que agrupa varias máquinas virtuales) tiene su propia red local, donde las máquinas virtuales tienen direcciones IP privadas de clase C en el rango 192.168.3.0/24, por lo que pueden comunicarse entre ellas. Además, cada *tenancy* tiene un enrutador virtual que permite a las máquinas dentro de la *tenancy* comunicarse con las máquinas fuera de su tenancy asegurando que los paquetes se envían a la máquina correcta. También proporcionan funciones propias de un NAT que posibilita asignar a las máquinas virtuales una

dirección IP visible fuera de la *tenancy*. En la nube administrada, esto se traduce en una dirección IP visible en la red JASMIN. En la nube no administrada, se traduce en una dirección IP pública.

Una funcionalidad muy interesante dentro de estos servicios cloud es el *Cluster as a Service* (CaaS), que permite el provisionamiento y mantenimientos de diversos tipos de clústeres gracias una simple interfaz web dentro del *JASMIN Cloud Portal*[16]. Uno de los tipos de clúster soportados es el clúster *Kubernetes*, tal y como ha sido descrito en el apartado 2.2.2. Con unos pocos *clicks* de ratón, un usuario puede tener a su disposición un clúster *Kubernetes* para desplegar el tipo de aplicación o servicio que desee, incluyendo por ejemplo las soluciones de acceso a datos climáticos recogidas en este trabajo. Actualmente esta funcionalidad se encuentra en período de pruebas y solo puede ser desplegada en la nube no administrada, con todas las restricciones que ello supone, por lo que el CaaS no ha sido utilizado en las tareas recogidas en este trabajo.

Todos los despliegues y pruebas se han realizado en una máquina virtual en una *tenancy* de la *External Cloud* de JASMIN, que ha actuado como un clúster *Kubernetes* de un único nodo, de ahí la importancia que ha tenido esta infraestructura en el trabajo y la necesidad de explicar su arquitectura y funcionamiento.

[16] https://cloud.jasmin.ac.uk/

Capítulo 4: El Programa Copernicus

"El programa Copernicus pone a disposición de los ciudadanos, las autoridades públicas, los responsables políticos, los científicos, los emprendedores y las empresas, un extenso mundo de información y conocimiento sobre nuestro planeta, de manera completa, abierta y gratuita.

Copernicus es una iniciativa inherentemente multidisciplinar y genuinamente europea, que aúna a comunidades de todo el espectro científico, desde la información geográfica al medio ambiente. Ofrece servicios operativos para distintas aplicaciones, como por ejemplo la monitorización del hielo marino del Ártico, la respuesta a emergencias, la detección de vertidos de petróleo, o el seguimiento de la expansión urbana. Los servicios Copernicus apoyan una amplia gama de aplicaciones medioambientales y de seguridad, incluida la vigilancia del cambio climático, el desarrollo sostenible, el transporte y la movilidad, la planificación regional y local, la vigilancia marítima, la agricultura y la salud." [18]

El programa Copernicus se apoya en una familia de satélites llamados *Sentinel*, propiedad de la Unión Europea, de infraestructuras espaciales ya existentes y de un gran número de sistemas de medición *in situ* puestos a disposición del programa por los Estados miembros. Estos pueden ser sensores colocados en las orillas de los ríos, flotando en el océano, o instalados en globos meteorológicos, barcos, etc. Las medidas *in situ* se utilizan para calibrar, verificar y complementar la información proporcionada por los satélites, lo que los hace esenciales para suministrar datos fiables y consistentes en el tiempo.
Los servicios Copernicus transforman los datos de satélite e *in situ* en información de valor añadido gracias al procesamiento y el análisis de los mismos, a su integración con otras fuentes y a la validación de los resultados. Las series de datos que se remontan años y décadas atrás se pueden consultar y comparar, lo que garantiza el seguimiento y la detección de cambios en las tendencias.
Estas actividades de valor añadido se han organizado en torno a seis ejes temáticos de servicios Copernicus [19], que se listan en la Tabla 2:

	Copernicus Atmosphere Monitoring Service (CEMS)	Vigilancia atmosférica
	Copernicus Marine Environment Monitoring Service (CMEMS)	Vigilancia medioambiental marina
	Copernicus Climate Change Service (C3S)	Cambio climático
	Copernicus Land Monitoring Service	Vigilancia terrestre
	Copernicus Emergency Management Service	Gestión de emergencias
	Copernicus Security Service	Seguridad

Tabla 2 - Servicios Copernicus

Las tareas realizadas en este trabajo forman parte de proyectos vinculados al servicio C3S, y más concretamente al proyecto *Climate Projections for the Copernicus Climate Data Store* (CP4CDS).

4.1 Copernicus Climate Change Service (C3S)

El objetivo de C3S[17] [20] es dar apoyo a las políticas de la Unión Europea para la adaptación y mitigación del cambio climático. Proporciona información sobre el pasado, presente y futuro del clima en Europa y el resto del mundo a sus usuarios, que incluyen científicos, consultores, medios de información y público en general.
C3S es dirigido por el *European Centre for Medium-range Weather Forecast* (ECMWF), una organización intergubernamental independiente, y la mayoría de sus servicios y productos son implementados por unas 200 compañías y organizaciones europeas seleccionadas en base a licitaciones públicas, que son convocatorias en las que se invita a dichos agentes a ofrecer sus soluciones para proveer bienes o servicios bajo un contrato.

Para llevar a cabo su misión, el servicio C3S se organiza en distintos elementos, por ejemplo el *Sistema de Información Sectorial* (SIS) y un *Sistema para la Evaluación y Control de Calidad* (EQC), pero el elemento más importante es el Almacén de Datos Climáticos,

[17] https://climate.copernicus.eu/

conocido como *Climate Data Store* (CDS), que actúa como una plataforma de acceso único a todos los datos del C3S. Debido a su importancia en este trabajo, se incluye una explicación detallada a continuación.

4.1.1 Climate Data Store

Como ya se ha mencionado, la misión fundamental del *Copernicus Climate Change Service (C3S)* es dar acceso a datos climáticos, y para ello el CDS es la piedra angular de la infraestructura C3S. El *Climate Data Store* [21] contribuye a la provisión de análisis, variables esenciales, proyecciones climáticas e indicadores en una escala temporal y espacial relevantes para las estrategias de actuación frente al cambio climático de diversos sectores sociales.
Los requerimientos que se establecieron para el diseño del CDS son:

- Proporcionar una visión central y holística de toda la información disponible para el C3S
- Proporcionar un acceso consistente y fluido a los repositorios de datos existentes, distribuidos en múltiples proveedores
- Proporcionar un catálogo de todos los datos y productos disponibles
- Proporcionar información sobre la calidad de todos los datos y productos
- Proporcionar acceso a herramientas software que permita a los usuarios procesar los datos
- Proporcionar un sistema que esté íntegramente monitorizado en términos de uso, disponibilidad y tiempos de respuesta

A esta lista se debería añadir el requerimiento de la Comisión Europea de que el CDS debería basarse en la infraestructura ya existente y que los datos deben permanecer en los proveedores, por ello el CDS fue desarrollado como un sistema distribuido.
El CDS también soporta múltiples formatos de datos (netCDF-CF, WMO-GRIB…) que son almacenados utilizando diferentes tecnologías (bases de datos relacionales, sistema de ficheros, archivos en cinta…) y accesibles a través de diferentes protocolos (servicios web, APIs...). El CDS pretende actuar como una única pasarela de acceso y procesado de todos estos datos.
A continuación, se comentan brevemente diversos elementos de la infraestructura del *CDS* (ver Figura 13):

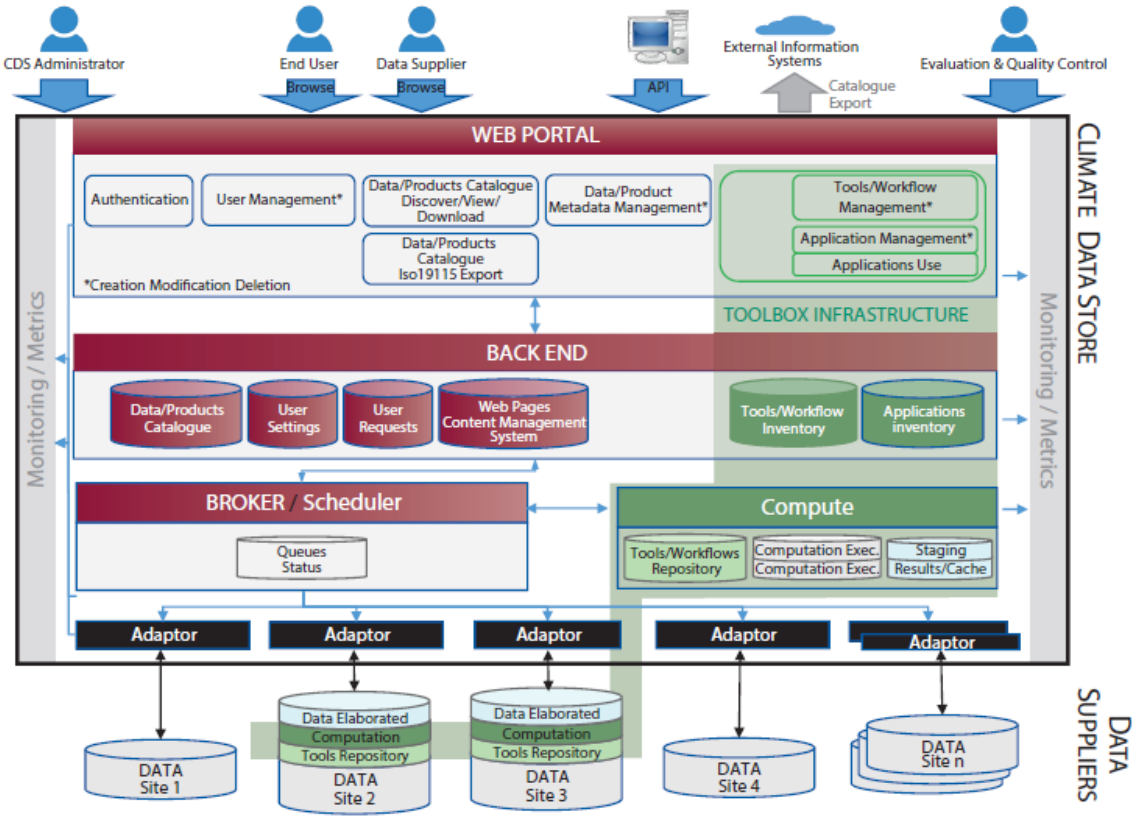

Figura 13 - Infraestructura del Climate Data Store (fuente: [21])

***Web Portal*:** el portal web actúa como punto único de entrada para el descubrimiento y manipulación de los datos disponible en el CDS. Permite a los usuarios buscar datos en el CDS, descargarlos, invocar herramientas del *Toolbox* para realizar operaciones sobre ellos y visualizar o descargar los resultados.

***Broker*:** este componente organiza, programa y redirige los datos y peticiones de procesado al repositorio de datos o capa de computación más conveniente a través de una serie de adaptadores. También implementa un módulo de calidad de servicio que encola las peticiones y programa su ejecución en base a una serie de reglas tomando en cuenta diferentes parámetros como el perfil de usuario, el tipo de petición o el coste de la misma (en términos de coste de CPU, volumen de datos, etc.). Este módulo es necesario para proteger al CDS de ataques de denegación de servicio (DoS); las peticiones de mayor volumen tienen asignadas una menor prioridad y el número de peticiones simultáneas por usuario está limitado. Así se garantiza que el sistema permanezca disponible.

***Adaptors*:** el papel de los adaptadores es traducir los datos y peticiones de procesado del *broker* en peticiones que puedan ser comprendidas por la infraestructura de cada proveedor de datos. Se espera que cada proveedor en la medida de lo posible proporcione acceso de acuerdo a algún estándar acordado (OpenDAP, ESGF, WPS, REST API…).

***Backend*:** el backend contiene varias bases de datos; desde el catálogo de datos y productos disponibles en el CDS al catálogo de herramientas disponibles pasando por información de control de calidad o información relacionada con el usuario (como el estado de sus peticiones o las licencias de uso de los datos).

Compute: la capa de computación es la encargada de realizar operaciones de procesado sobre una combinación de datos provenientes de múltiples repositorios remotos. Estas operaciones pueden realizarse en los propios repositorios cuando sea posible o en el entorno *cloud* que sostiene el CDS.

4.1.2 Climate Projections for the Climate Data Store

Este proyecto del C3S se creó para el aprovisionamiento del CDS con datos provenientes de simulaciones de proyecciones globales climáticas. Dichas simulaciones se realizan bajo la iniciativa *Coupled-Models Intercomparision Project* (CMIP) parte del *World Climate Research Program* (WCRP) de la *World Meteorology Organizations* (WMO). CMIP tiene como objetivo obtener simulaciones para los distintos informes sobre el cambio climático del *Inter-gubernamental Panel for Climate Change* (IPCC). Concretamente las simulaciones suministradas por CP4CDS corresponden a las simulaciones del Fifth Assessment Report del IPCC y que se denominan CMIP5.

Los objetivos de CP4CDS se pueden resumir en dos puntos:

1. Proporcionar acceso a datos procedentes de CMIP5 para el CDS a través de la arquitectura ESGF y de sus *Data Nodes e Index Nodes*. Al mismo tiempo, proporcionar capacidades de computación para el procesado de los datos utilizando *Web Processing Services* (ver Figuras 14 y 15).

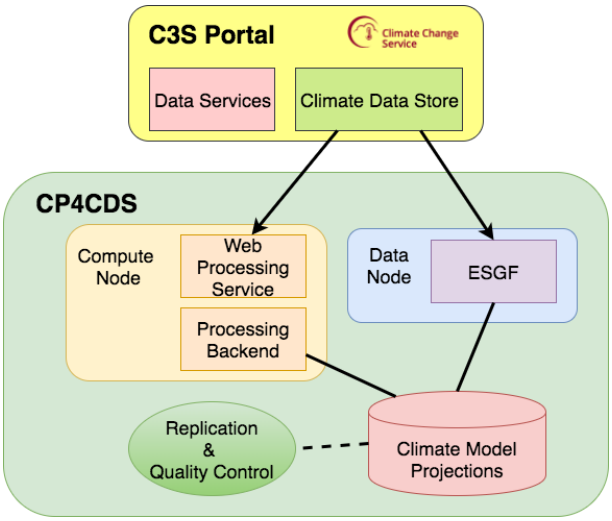

Figura 14 - CP4CDS: Acceso y procesado de datos para el CDS (fuente: [22])

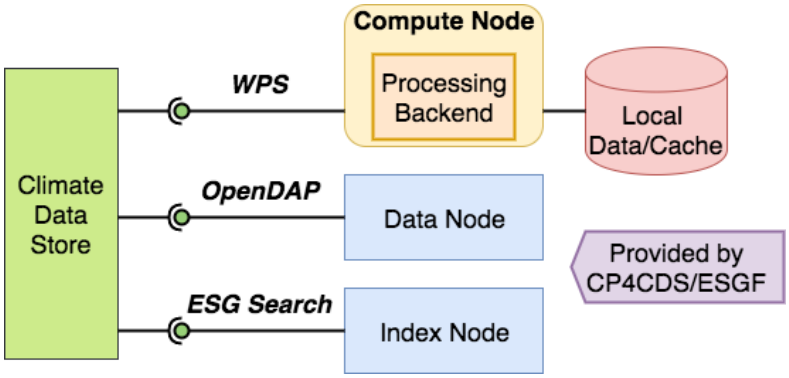

Figura 15 - CP4CDS: Interfaces para el CDS (fuente: [22])

2. Una infraestructura geográficamente distribuida que balancea la carga entre tres nodos alojados en los tres centros de investigación meteorológica líderes en Europa que participan en el proyecto: CEDA en Reino Unido, IPSL en Francia y DKRZ en Alemania. Los tres nodos tienen los mismos datos replicados y la misma pila de software desplegada. CEDA aloja el nodo principal e IPSL y DKRZ asumen el control cuando es necesario (ver Figura 16).

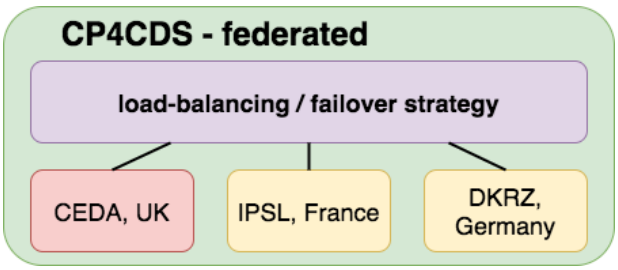

Figura 16 - Infraestructura federada para CP4CDS (fuente: [22])

Las tareas realizadas en este trabajo se alinean con el primero de estos objetivos, como se verá más adelante.

Capítulo 5: Implementación Técnica

El objetivo de este TFG es demostrar el escalado de los servicios de acceso a datos climáticos haciendo uso de *software* de orquestación de contenedores.
Esta evaluación forma parte de las actividades del proyecto CP4CDS titulado "*Provision of support to Earth System Grid Federation (ESGF) node in Europe"* [23] (Provisión de soporte a un nodo ESGF en Europa). Su objetivo es facilitar el acceso y procesado de proyecciones climáticas existentes procedentes de la iniciativa CMIP5 a través de una interfaz diseñada para el *Climate Data Store*.

Este lote de tareas, sirve para demostrar el escalado del servicio con nodos de datos y computación que satisfagan la demanda. Además, implica demostrar la capacidad de escalado dentro de un mismo centro que permite el uso de las tecnologías mencionadas en este TFG, especialmente la orquestación de contenedores con *Kubernetes*.

En concreto, las tareas realizadas han sido las siguientes:

1) Adaptación de una *Chart* de *Helm* ya existente con todos los componentes de un nodo *ESGF* en forma de contenedores *Docker* para que incluya el autoescalado de los servidores *THREDDS* de un *Data Node.*
2) Creación de una *Chart* de *Helm* para el despliegue de *Birdhouse,* como servicio de computación consistente en *Web Processing Services*, incluyendo el autoescalado del *front-end*.
3) Recogida de métricas que demuestren el autoescalado en las tareas 1) y 2).

El primer paso necesario para cumplir con dichas tareas es el despliegue de un clúster Kubernetes. Para las pruebas realizadas como parte de este TFG, se ha desplegado un clúster de un único nodo consistente en una máquina virtual con sistema operativo CentOS 7 dentro de la *External Cloud* de JASMIN, para lo cual se ha utilizado *Rancher*.
Rancher [24] es un *software* que facilita el despliegue y la gestión de múltiples clústeres *Kubernetes*, ya sea en una infraestructura propia o en infraestructuras de proveedores de servicios de nube pública. *Rancher* dispone de multitud de herramientas para la administración de clústeres en producción (gestión centralizada de la seguridad, integración de la autenticación con servicios de terceras partes, monitorización avanzada…), no obstante estas características carecen de relevancia para este trabajo, ya que las tareas han sido realizadas siempre en un entorno de desarrollo y el uso de *Rancher* se ha limitado exclusivamente al despliegue del clúster con uno de sus componentes, *Rancher Kubernetes Engine* (RKE)[18].
Todo lo que RKE precisa para el despliegue de un nodo en un clúster *Kubernetes* es su dirección IP y especificar de qué tipo de nodo se trata (un mismo nodo puede ser de varios tipos), en función de los componentes de la arquitectura de *Kubernetes* que se pretendan instalar en él. Estos tipos son tres: *etcd* y *control plane*, para componentes maestros, y *worker*, para componentes de nodo. También es necesario proporcionar un usuario con acceso *SSH* a los mismos mediante una pareja de claves pública y privada y que cuente con los permisos necesarios para ejecutar contenedores Docker. Adicionalmente, permite añadir otros parámetros de configuración como las imágenes utilizadas para los componentes,

[18] https://rancher.com/docs/rke/latest/en/

parámetros del clúster como el rango de direcciones *IP* asignadas a los *Pods* o a los *Services* o incluso incluir el despliegue de un *Ingress Controller*.
Los nodos, por su parte, deben cumplir una serie de requisitos [25]:

- Sistema Operativo con Docker instalado:
    - Ubuntu 16 o Ubuntu 18
    - Red Hat Enterprise Linux (RHEL)/CentOS 7
    - RancherOS
    - Windows Server 2019 (experimental)
- Tener instalada una versión de *OpenSSH* superior a 6.4, que soporte la creación de túneles SSH para el transporte de información cifrada.
- Acceso a una serie de puertos TCP/UDP en función del tipo de nodo. Uno de ellos, por ejemplo, es el puerto 6443/TCP necesario para las comunicaciones con el *apiserver* de *Kubernetes;* ver [3] para la lista completa. Como ya se ha mencionado, en la *External Cloud* de JASMIN la seguridad es responsabilidad del usuario por lo que estos puertos han tenido que ser abiertos en las máquinas manualmente (mediante *firewalld* o *iptables*).

Una vez se cumplan estos requisitos basta con descargar un binario[19], ejecutarlo con el comando `rke up` y completar el formulario que aparece en consola con los parámetros mencionados anteriormente. Otra opción es crear un fichero YAML con dichos parámetros, como el utilizado en el despliegue del clúster para el desarrollo de este TFG (ver Figura 17):

[19] https://rancher.com/docs/rke/latest/en/installation/

```
---
nodes:
    - address: 192.171.139.109
      port: 22
      user: pcelaya
      role:
        - controlplane
        - etcd
        - worker
      ssh_key_path: /home/pcelaya/.ssh/id_rsa_jasmin

# Enable use of SSH agent to use SSH private keys with passphrase
# This requires the environment `SSH_AUTH_SOCK` configured pointing
# to your SSH agent which has the private key added
ssh_agent_auth: true

cluster_name: mycluster

system_images:
    kubernetes: rancher/hyperkube:v1.15.3-rancher2
    etcd: rancher/coreos-etcd:v3.2.24
    kubedns: rancher/k8s-dns-kube-dns-amd64:1.14.8
services:
    kube-api:
      # IP range for any services created on Kubernetes
      # This must match the service_cluster_ip_range in kube-controller
      service_cluster_ip_range: 10.43.0.0/16
      # Expose a different port range for NodePort services
      service_node_port_range: 30000-32767
    kube-controller:
      # CIDR pool used to assign IP addresses to pods in the cluster
      cluster_cidr: 10.42.0.0/16
      # IP range for any services created on Kubernetes
      # This must match the service_cluster_ip_range in kube-api
      service_cluster_ip_range: 10.43.0.0/16
ingress:
    provider: nginx
```

Figura 17 - Fichero `cluster.yaml` con los parámetros de configuración para RKE

Después de que RKE despliegue el clúster, se genera un fichero llamado `kube_config_cluster.yml`, con la información necesaria para poder interactuar con el clúster *Kubernetes* haciendo uso del programa de línea de comandos `kubectl`.

5.1 Configuración del autoescalado de servidores THREDDS para un nodo ESGF

Existe un proyecto colaborativo denominado *"ESGF Docker"*[20] que proporciona los componentes de la arquitectura ESGF en forma de contenedores *Docker*[21]. Estos

[20] https://esgf.github.io/esgf-docker/
[21] https://hub.docker.com/u/esgfhub

componentes pueden ser desplegados en un clúster *Kubernetes*, concretamente en forma de una *Chart* de *Helm*.

En el repositorio *Github* del proyecto[22] se puede descargar todo lo necesario para el despliegue. Dicho repositorio consta de una serie de *scripts* (que también se ejecutan dentro de un contenedor) para la configuración del nodo (generación de contraseñas y claves para el acceso al nodo, creación de certificados de prueba autofirmados…) y para realizar el despliegue utilizando diferentes métodos. Como ya se ha visto, todo despliegue en *Kubernetes* consta de múltiples componentes (*Deployment*, *Volumes*, *Services, Ingress*…) y un nodo ESGF consta a su vez de multitud de elementos, por lo que se intuyen las facilidades que otorga utilizar una *Chart* de *Helm* en cuestiones de flexibilidad en el despliegue y portabilidad; también es fácil imaginar la dimensión del árbol de directorios y ficheros que conforman la *Chart*, por lo que se omite entrar en detalle en este apartado.

Para incorporar la capacidad de autoescalado de los servidores THREDDS, se añade en el directorio correspondiente el *manifest* de un *HorizontalPodAutoscaler* en forma de plantilla de tal manera que algunos de sus parámetros puedan ser configurados como es habitual al utilizar *Helm*.

[22] https://github.com/ESGF/esgf-docker

```
---
{{ if .Values.tds.hpa.enabled }}
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: "{{ template "fullname" . }}-tds"
  labels:
{{ include "default-labels" . | indent 4 }}
    component: tds
spec:
  minReplicas: {{ .Values.tds.hpa.minReplicas }}
  {{ if lt .Values.tds.hpa.maxReplicas .Values.tds.replicas }}
  maxReplicas: {{ .Values.tds.replicas }}
  {{ else }}
  maxReplicas: {{ .Values.tds.hpa.maxReplicas }}
  {{ end }}
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: "{{ template "fullname" . }}-tds"
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: {{ .Values.tds.hpa.cpuUtilization }}
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization: {{ .Values.tds.hpa.memUtilization }}
{{ end }}
```

Figura 18 - Template `hpa.yml` para el autoescalado horizontal de servidores THREDDS

Como se puede apreciar en la Figura 18, el usuario que despliegue esta *Chart* de *Helm* puede especificar (bien de forma explícita con la opción `--set` del cliente `helm`, bien en un fichero tipo `values.yaml`) si quiere activar el autoescalado (`.Values.tds.hpa.enabled`), el número mínimo y máximo de réplicas (`.Values.tds.hpa.minReplicas` y (`.Values.tds.hpa.minReplicas`) y los valores límite de CPU) y memoria (`.Values.tds.hpa.cpuUtilization` y `.Values.tds.hpa.memUtilization`) para que se produzca el autoescalado de los *Pods* TDS[23]. Estos valores se expresan como porcentaje de los recursos que se asignan al *Pod* en su despliegue y se calculan como la media aritmética de todos los *Pods* que estén ejecutándose.

[23] THREDDS Data Server

Como aportación al repositorio de "*ESGF Docker"* y siguiendo la guía de contribución[24], se ha creado una *issue*[25] en *Github* con el autoescalado de los servidores THREDDS como propuesta de mejora. Posteriormente se ha creado una *Pull Request*[26] asociada a dicha *issue* con los cambios mencionados para que sean incorporados al repositorio. No obstante, dicha *Pull Request* ha sido etiquetada como "*Work In Progress" (WIP)* para explorar posibles mejoras.

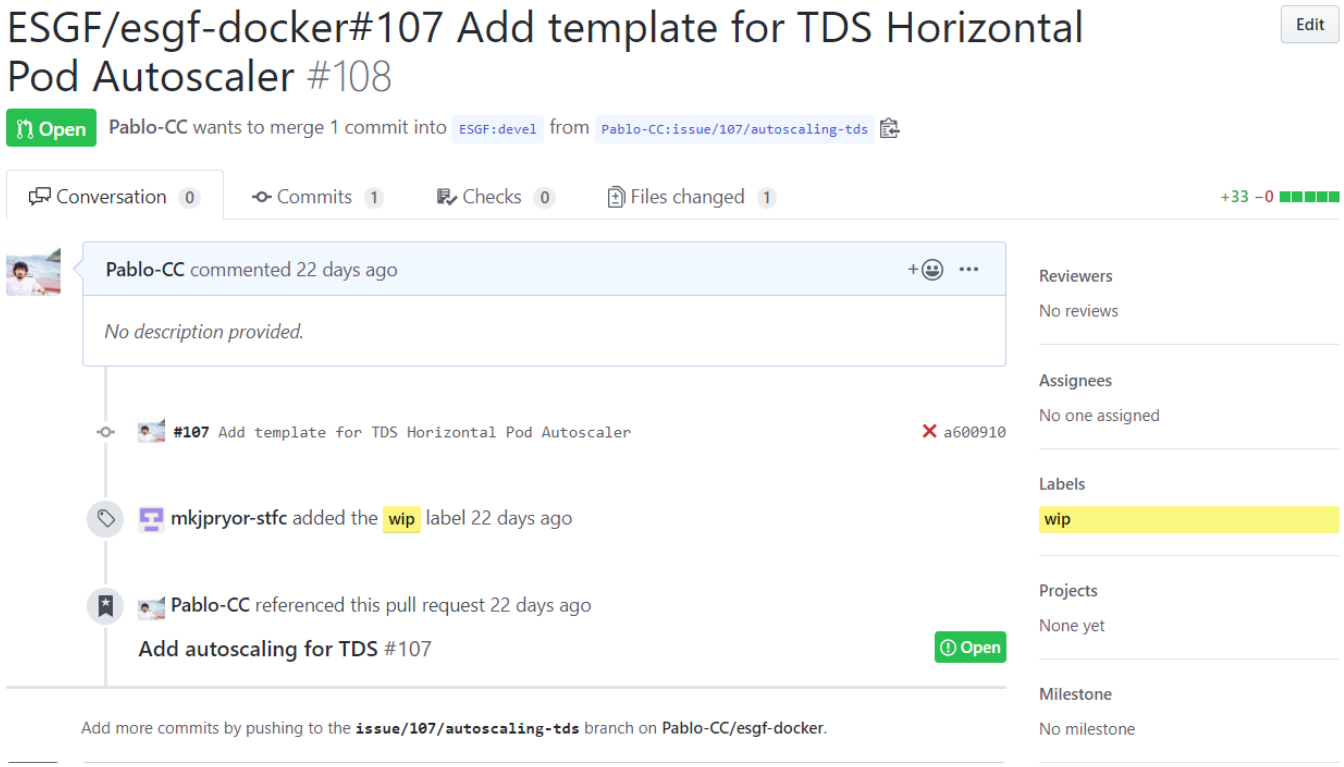

Figura 19 - Pull Request creada en el repositorio oficial del proyecto "ESGF Docker"[27]

5.2 Despliegue de Birdhouse mediante un *Chart de Helm*

Birdhouse está diseñado para ser instalado fácilmente, puede utilizar para el despliegue *Buildout* (herramienta de automatización de despliegues basada en ficheros de configuración), *Ansible* (herramienta *DevOps* para configuración y administración de múltiples nodos en remoto) o contenedores Docker.

Como ya se ha comentado, *Birdhouse* consta de múltiples componentes. Es por esto que, de la misma forma que el software de un nodo ESGF, es ideal para ser adaptado para su despliegue en un clúster *Kubernetes* con *Helm*.

Las tareas llevadas a cabo relacionadas con *Birdhouse* han sido el diseño de una *Chart* de *Helm* para el despliegue de algunos de los componentes de *Birdhouse,* en forma de contenedores *Docker,* en un *clúster Kubernetes*. Únicamente se han incluido unos pocos componentes (concretamente *Phoenix* como *front-end y como* WPS del *back-end Emu, Hummingbird y Flying Pigeon*) ya que esta tarea simplemente sirve como prueba de concepto

[24] https://esgf.github.io/esgf-docker/developer/contributing/
[25] Unidad de trabajo creada en *Github* que describe un problema o una posible mejora en un proyecto
[26] Propuesta para integrar una serie de cambios o propuestas en un proyecto de *Github*
[27] https://github.com/ESGF/esgf-docker/pull/108

para la mostrar la conveniencia de utilizar *Helm* para desplegar *Birdhouse* e incluir el autoescalado del *front-end* (en este caso, *Phoenix*).

En la *Chart* creada, disponible en *Github*[28], cada uno de los componentes consta de cuatro plantillas de *Helm*:

- `deployment.yml`
- `service.yml`
- `ingress.yml`
- `configMap.yml`

Adicionalmente, *Phoenix* incluye una plantilla (`hpa.yml`) para su autoescalado horizontal, muy similar a la de la Figura 18 para los servidores THREDDS.

En lo que se refiere al *Deployment* todos los componentes son bastante simples. Constan de *Pods* (tantas réplicas como especifique el usuario a la hora de desplegar con *Helm*) que a su vez constan de un único contenedor. La configuración de los contenedores se realiza a través de la definición de ciertas variables de entorno (para especificar los puertos TCP y el nombre de dominio) y de un *Volume* del tipo *configMap,* que como ya se ha mencionado en el capítulo 2.2.1 son objetos que sirven para definir parámetros de configuración de las aplicaciones. Los componentes de *Birdhouse* se instalan en el contenedor por medio de *Buildout,* un programa para la instalación de aplicaciones que lee parámetros procedentes de ficheros, como los creados en este caso con los *configMap*.
Para el caso de *Phoenix*, la Figura 20 muestra el fichero de configuración `custom.cfg` ya desplegado.

```
[buildout]
extends=profiles/docker.cfg
[settings]
hostname = ${environment:HOSTNAME}
#These ports have to be
http-port = ${environment:HTTP_PORT}
https-port = ${environment:HTTPS_PORT}
mongodb-host = mongodb
mongodb-port = 27017
twitcher-host = localhost
twitcher-port = ${:https-port}
twitcher-url = https://${:twitcher-host}:${:twitcher-port}
twitcher-delegate = false
twitcher-workdir = ${settings:prefix}/var/lib/twitcher
#Use setPasswd.py to generate your own password (>6 characters)
#Default password: qwerty
phoenix-password =
sha256:ee4886f80459:2a3e88c8d606431c5a124fc08b747acd5fdb49033203a2bdfca1f
08d40f22818
```

Figura 20 - Fichero `custom.cfg` con los parámetros de configuración de Phoenix

[28] https://github.com/Pablo-CC/birdhouse-helm/

Por último, queda por tratar la exposición de los servicios. *Phoenix* y los WPS se comunican entre sí a través de un *Service* de tipo *ClusterIP*, accesible solamente dentro del clúster *Kubernetes*. *Phoenix* se expone al exterior a través del *Ingress Controller* del clúster, de ahí la necesidad de crear un objeto *Ingress* que redirija al *Service* mencionado. Además, en ocasiones los resultados de un proceso se entregan en un fichero al usuario, pasándole a este una URL para la descarga del mismo; es por ello que los propios WPS del *back-end* también son expuestos directamente al usuario con un *Ingress*. Así, en el intercambio de mensajes HTTP de este despliegue de *Birdhouse* se utilizarán dos tipos de URLs:

- Para la comunicación interna entre *Phoenix* y los WPS del *back-end*: se utilizan URLs en las que el nombre de dominio es del tipo que resuelve el *cluster DNS* de *Kubernetes* para un *Service ClusterIP*.

  `<nombre Service>.<espacio de nombres>`
  (Ej: `emu.birdhouse`)

- Para la comunicación directa entre el usuario y los componentes de *Birdhouse*: tanto el *Ingress Controller* como las propias aplicaciones detrás de los WPS desplegados necesitan un nombre de dominio válido que sea accesible desde el exterior (una dirección IP no se considera un nombre de dominio válido). En un entorno de producción este nombre de dominio estaría registrado en un registro DNS por la entidad que gestiona el servicio (por ejemplo `phoenix.ceda.ac.uk`), sin embargo para el entorno de desarrollo de este TFG se ha utilizado *nip.io*[29], que es un servicio *DNS wildcard* gratuito que permite registrar múltiples nombres de dominio bajo la misma dirección IP y además incluir la propia dirección IP en el nombre, de tal forma que la traducción es automática sin necesidad de crear un registro DNS. Las URLs para el acceso directo del usuario serían de la forma:

  `<nombre componente Birdhouse>.< IP Ingress Controller>.nip.io`
  (Ej: `emu.192.171.139.109.nip.io`)

5.3 Demostración del autoescalado con *Kubernetes*

Para probar la eficiencia de *Kubernetes* en el autoescalado horizontal de los servicios desplegados en las tareas anteriores, se han recogido una serie de métricas de uso en el clúster *Kubernetes* para después poder representarlas gráficamente.
Para la recogida de métricas se ha utilizado el *software* de código abierto *Prometheus*[30], un sistema de monitorización que se comunica con la API de *Kubernetes* para registrar las métricas que el usuario solicite y que utiliza bases de datos de series temporales para almacenarlas. Utiliza un lenguaje de consultas propio, *PromQL*, *y* expone una API HTTP para la integración con otros servicios que permite crear nuevas solicitudes, consultar métricas ya registradas, configurar alertas, etc. Uno de los servicios con los que se integra habitualmente es *Grafana*[31].

[29] https://nip.io/
[30] https://prometheus.io/docs/introduction/overview/
[31] https://grafana.com/docs/

Grafana es un *software* de visualización de datos, también de código abierto, que proporciona una interfaz gráfica en formato web para ser utilizada a través del navegador. Como se ha comentado, se puede integrar con *Prometheus* para que lance consultas en lenguaje *PromQL*, recoja los datos y los represente gráficamente. En este TFG se ha hecho uso de *Grafana* para la creación de paneles (llamados *dashboards*) que agrupen las consultas relevantes a cada servicio y separe las métricas en función del servicio que se esté monitorizando. Posteriormente, se han exportado los datos en ficheros CSV para su procesado con librerías del lenguaje de programación *Python*, que dan mayor flexibilidad en la representación y formateado de las gráficas.
Concretamente se ha hecho uso de dos librerías: *pandas*[32] *y matplotlib*[33]. La primera es una librería para el análisis de datos que cuenta con unas estructuras de datos (denominadas *DataFrames)* necesarias para limpiar los datos en bruto (los provenientes de los ficheros CSV) y que sean aptos para el análisis y, en este caso, su representación gráfica. La segunda de las librerías, *matplotlib*, es el estándar de facto en *Python* para la creación de todo tipo de gráficas en múltiples formatos.

Las métricas que han sido recogidas, tanto para los servidores THREDDS como para *Phoenix,* son las siguientes:

- Uso de CPU de los *Pods*
- Uso de memoria RAM de los *Pods*
- Peticiones por segundo al *Ingress Controller*
- Número de réplicas en el *Deployment*

Para la métrica de las peticiones por segundo en el caso de los servidores THREDDS, es necesaria una modificación en la configuración del nodo ESGF, que se explica a continuación. Algo que no se ha mencionado previamente, es que el nodo ESGF puede precisar autenticación del cliente con TLS, que es una prestación que *Kubernetes* aún no soporta, y por ello en el despliegue se incluye un contenedor *proxy* que permite esta funcionalidad. Así, solamente hay un objeto *Ingress* configurado en modo *SSL passthrough* (lo que significa que el tráfico atraviesa cifrado el *Ingress Controller*) apuntando a dicho contenedor *proxy* que realiza la terminación del cifrado antes de redirigir el tráfico a las aplicaciones finales (por ejemplo, el servidor THREDDS); este comportamiento del contenedor *proxy* se conoce como *SSL Offloading*. Al ir el tráfico cifrado, el *Ingress Controller* no puede reportar métricas al respecto. En las pruebas realizadas para esta tarea, por no ser necesaria la autenticación del cliente, se ha modificado manualmente el *Ingress* para que no utilice el modo *SSL Passthrough* y se han proporcionado los certificados al *Ingress Controller* para que sea él quien realice el *SSL Offloading* y todo el tráfico del nodo ESGF sea HTTP.

Por último, para generar la carga de trabajo sobre los servidores (tanto THREDDS como *Phoenix*) y provocar el autoescalado, se ha utilizado la herramienta de línea de comandos *siege*[34]. Esta es una herramienta utilizada generalmente por desarrolladores para realizar pruebas sobre servidores web, enviando un gran número de peticiones HTTP y HTTPS de manera simultánea (es un *software multithread*) para estresar al servidor y obtener datos de

[32] https://pandas.pydata.org/
[33] https://matplotlib.org/
[34] https://www.joedog.org/siege-home/

rendimiento con el objetivo de optimizar la configuración. En este caso particular, se ha utilizado para simular tráfico real de múltiples clientes generando una carga continua durante 10 minutos sobre una serie de URLs aleatorias.

A continuación, se incluyen las gráficas generadas en *Python* a partir de las métricas recogidas con *Prometheus* y exportadas de los paneles de *Grafana*.
En las Figuras 21, 22 y 23 se observan las métricas pertenecientes a los servidores THREDDS del nodo ESGF desplegado con *Helm*.

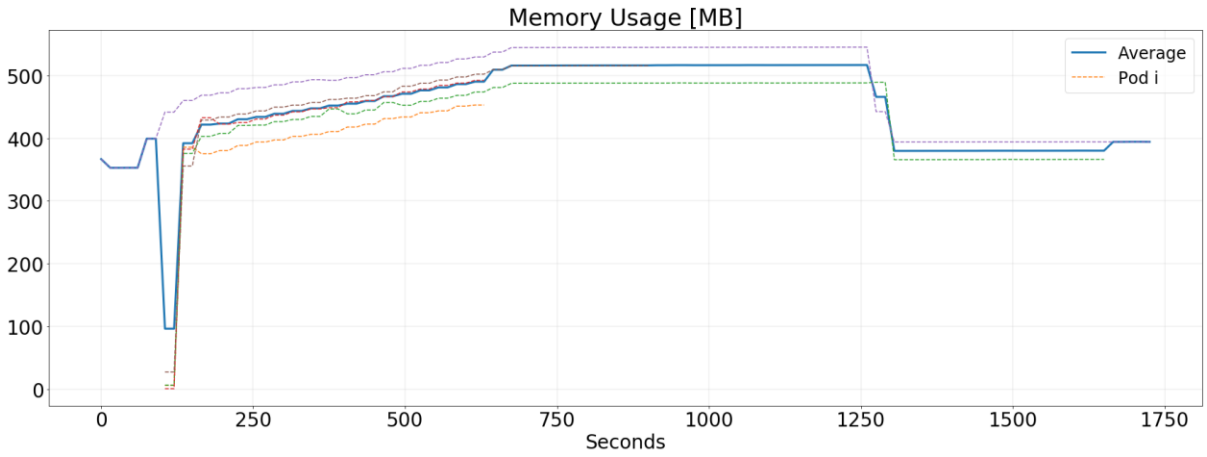

Figura 21 - Uso de memoria de los Pods THREDDS

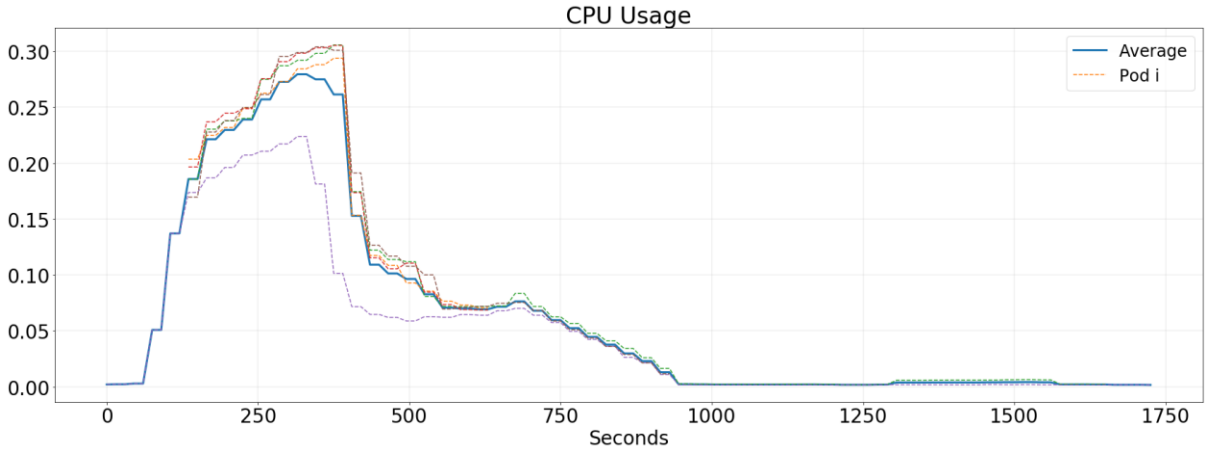

Figura 22 - Uso de CPU de los Pods THREDDS

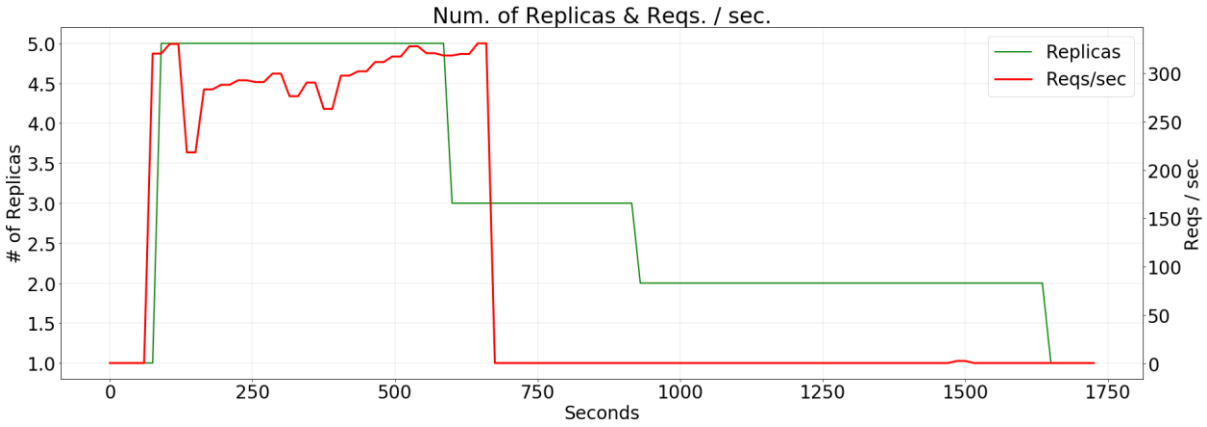

Figura 23 - Peticiones por segundo y número de réplicas durante la prueba

Para estas pruebas se han configurado valores muy bajos de los recursos de CPU y memoria en el HPA ya que simplemente se trata de una demostración. En un entorno de producción real, al dotar a los componentes de más recursos los valores serían considerablemente mayores.
En el caso de los servidores THREDDS los parámetros del HPA han sido:

- Réplicas mínimas: 1
- Réplicas máximas: 8
- Umbral de CPU: 160m (“160 milésimas de un core”)
- Umbral de memoria RAM: 450 MB

Esta prueba se ha realizado configurando *siege* para generar un total de aproximadamente 300 peticiones por segundo.
Se puede apreciar que ante una repentina subida de usuarios haciendo peticiones se produce un aumento abrupto en el uso de la CPU de los *Pods THREDDS* del nodo ESGF. Esto provoca que el *Horizontal Pod Autoscaler* responda inmediatamente (se puede ver en la Figura 23 que la latencia es muy baja) subiendo el número de réplicas a 5. Posteriormente, conforme las métricas se estabilizan el número de réplicas baja a 3 y espera un tiempo hasta volver a escalar por el mecanismo *anti thrashing* (ver apartado 2.2.3) momento en el cual el número de réplicas se reduce a 2. Finalmente, cuando la memoria RAM es liberada se vuelve a contar con una única réplica, como en origen.
La gráfica más importante es la de la Figura 23, pues muestra la relación directa entre una subida en el número de peticiones de los clientes y el autoescalado de los servicios. Esta gráfica demuestra la capacidad de escalado de un nodo de datos, que se produce de forma transparente a los usuarios puesto que estos se conectan a un *proxy* que les redirige a los servicios finales.

Por último, se presentan las gráficas para el caso del autoescalado de *Phoenix*, el *front-end* de *Birdhouse*, al realizar una prueba similar (Figuras 24, 25, y 26).

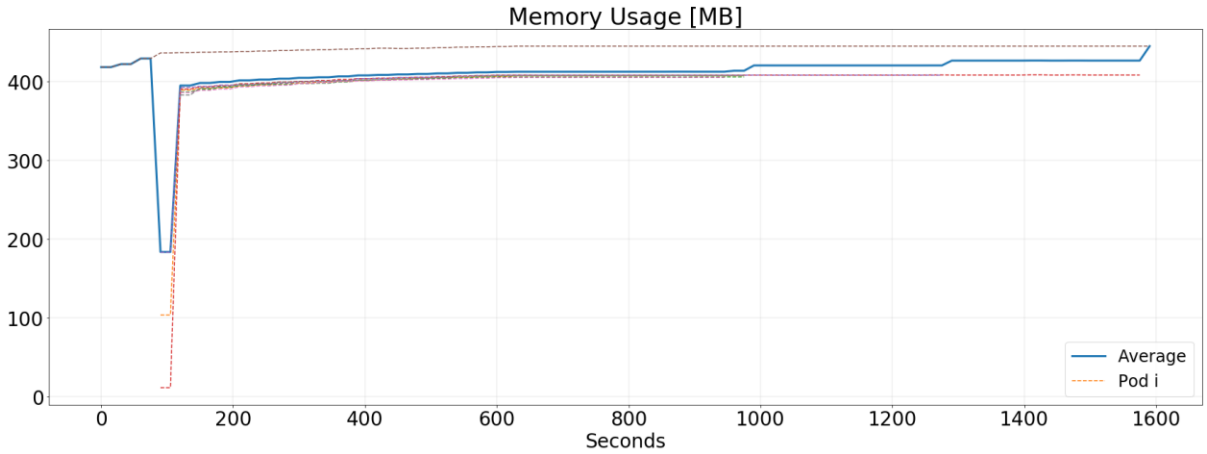

Figura 24 - Uso de memoria RAM de los Pods Phoenix

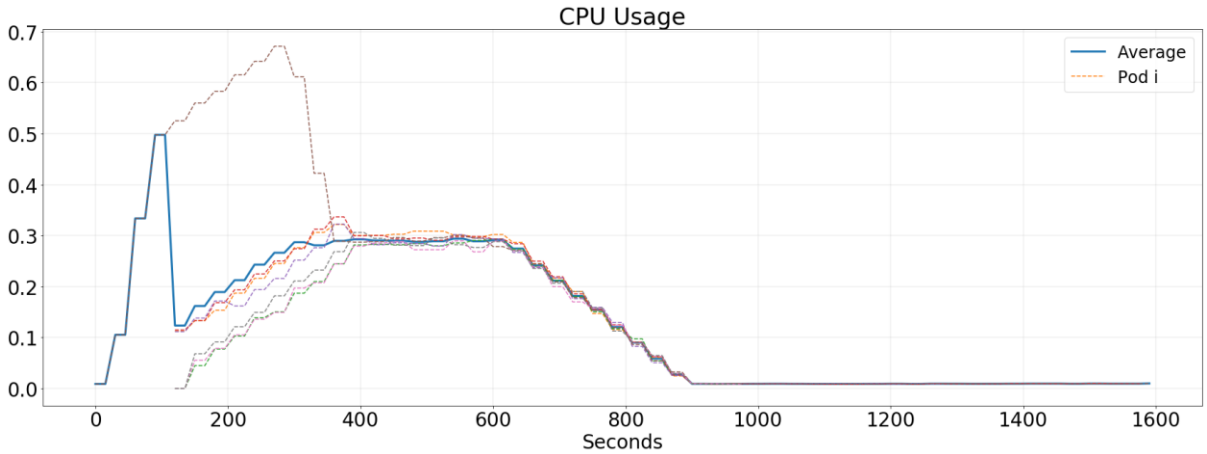

Figura 25 - Uso de CPU de los Pods Phoenix

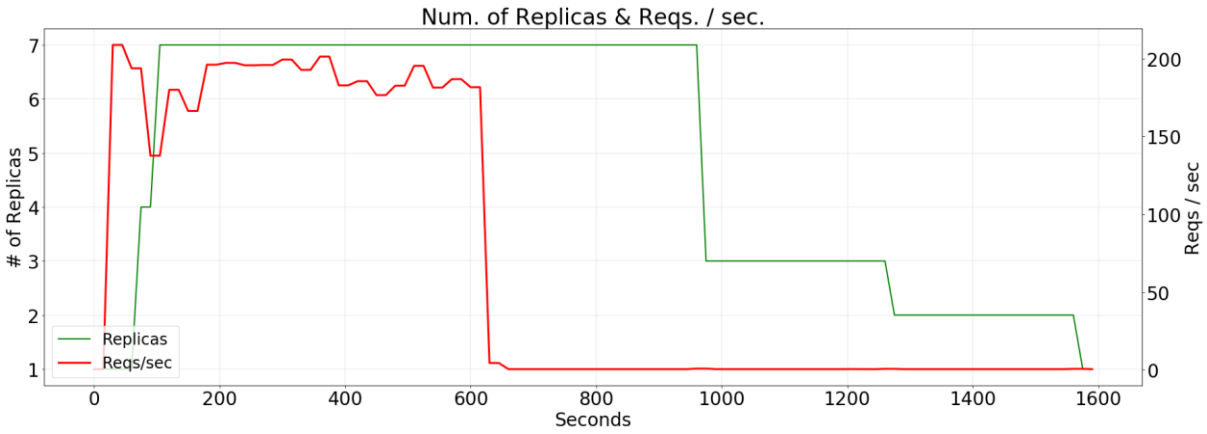

Figura 26 - Peticiones por segundo (línea roja) y número de réplicas (línea verde)

En este caso, los parámetros de configuración del HPA han sido:

- Réplicas mínimas: 1
- Réplicas máximas: 8
- Umbral de CPU: 300m (“300 milésimas de un core”)
- Umbral de memoria RAM: 450 MB

De nuevo, *siege* es ejecutado para realizar aproximadamente 300 peticiones HTTP por segundo.

El comportamiento observado es similar, salvo porque la latencia del autoescalado es mayor y los *Pods* tardan más tiempo en arrancar por lo que el *Pod* original absorbe más porcentaje del tráfico inicial.
Es importante señalar que, en ambas gráficas de uso de CPU, se observa que el equilibrio de la carga (que sigue un algoritmo de *round-robin*) funciona perfectamente y reparte las peticiones de manera equitativa, puesto que todos los *Pods* presentan un tiempo de CPU prácticamente idéntico.

5.4 Conclusiones

En este TFG se han planteado escenarios que demuestran la conveniencia de usar contenedores *software* y un sistema de orquestación como *Kubernetes*. Se han creado modelos nuevos para el despliegue de servicios de procesado de datos climáticos y se han configurado otros existentes para añadir nuevas funcionalidades. En los siguientes puntos se resume el éxito en el desempeño de este TFG, las aportaciones realizadas y el cumplimiento de los objetivos planteados:

- Se han probado las ventajas de utilizar *Kubernetes* para desarrollar *software* y gestionar servicios desplegados.
- Se han probado las facilidades que aporta *Helm* para el despliegue de aplicaciones complejas en un clúster *Kubernetes.*
- Se ha demostrado las capacidades para automatizar el escalado horizontal de servicios que proporciona *Kubernetes.*
- Se ha realizado una aportación a la comunidad de desarrollo de *software* de código abierto. Concretamente, se hace referencia en este punto a la *Pull Request* creada en el repositorio "*ESGF Docker*".
- Se ha generado un *deliverable* para un proyecto a nivel europeo. En este punto se hace referencia a la demostración del autoescalado de servidores THREDDS, cuyas gráficas irán recogidas en el informe final del proyecto CP4CDS.

5.5 Líneas futuras

Este trabajo ha servido para evaluar la arquitectura actual del despliegue del nodo ESGF para el CDS, pero la idea es continuar con nuevos despliegues incorporando más elementos y permitiendo una mayor flexibilidad. Por ejemplo, se podría estudiar la posibilidad de permitir despliegues parciales, ya que la *Chart* actual instala todos los componentes. Además, algunos servicios, como *GlobusFTP*, no cuentan con una implementación en *Docker,* por lo que se propone como línea futura desarrollar un contenedor que implemente su funcionalidad.

También se podría trabajar en la mejora de los contenedores que forman *Birdhouse*, ya que no cumplen enteramente con la filosofía detrás de esta tecnología y corren varios procesos simultáneamente; esto los haría más ligeros y mejoraría su velocidad de despliegue.

Otra línea futura que resultaría interesante investigar sería el uso de los *jobs* de *Kubernetes* (contenedores con un ciclo de vida muy corto que ejecutan funciones específicas y dan servicio a los elementos del clúster) para automatizar el proceso de publicación en un nodo.

Por último, este trabajo puede servir para aplicar todo lo aprendido al despliegue de la infraestructura del Servicio de Datos Climáticos del Grupo de Meteorología de Santander, que forma parte del catálogo de servicios científico-tecnológicos de la Universidad de Cantabria y el Consejo Superior de Investigaciones Científicas.
Esta experiencia permitirá adaptar esta tecnología de orquestación de contenedores en la nueva infraestructura del servicio, que permitirá ofrecer un entorno de acceso y análisis de datos para dar soporte al *Working Group I* del IPCC, en particular el acceso a datos del CMIP6 vía ESFG y su integración o coordinación con el almacenamiento a largo plazo del Data Distribution Center del IPCC.

Bibliografía

[1] M. Pryor, P. Kershaw, A. Iwi, S. Gardoll, Carsten Ebrecht, Luca Cinquini, “Highly Disponible ESGF Services for the Copernicus Climate Data Store”, 2018.

[2] R. Dua, A. R. Raja, and D. Kakadia, “Virtualization vs Containerization to Support PaaS” in *2014 IEEE International Conference on Cloud Engineering*, Boston, MA, USA, 2014, pp. 610–614.

[3] “What is Kubernetes.” [Online]. Disponible: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[4] “Understanding Kubernetes Objects.” [Online]. Disponible: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[5] “Ingress Controllers.” [Online]. Disponible: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

[6] “Kubernetes Architecture 101,” *https://www.aquasec.com*. [Online]. Disponible: https://www.aquasec.com/wiki/display/containers/Kubernetes+Architecture+101

[7] “Horizontal Pod Autoscaler.” [Online]. Disponible: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[8] “Helm.” [Online]. Disponible: https://helm.sh/

[9] L. Cinquini, D. Crichton, C. Mattman, J. Harney, G. Shipman, “The Earth System Grid Federation”, p. 10.

[10] ESGF – TracMeteo.” [Online]. Disponible: https://meteo.unican.es/trac/wiki/ESGF

[11] «Birdhouse Documentation. Release 0.5.0», p. 56.

[12] B. Pross, “OGC WPS 2.0.2 Interface Standard: Corrigendum 2”, p. 133, 2018.

[13] C. Ehbrecht, T. Landry, N. Hempelmann, D. Huard, y S. Kindermann, “Projects based on the Web Processing Service framework Birdhouse”, Int. Arch. Photogramm. Remote Sens. Spatial Inf. Sci., vol. XLII-4/W8, pp. 43-47, jul. 2018.

[14] N. Hempelmann, C. Ehbrecht, C. Álvarez-Castro, P. Brockman, W. Falk, “Web processing service for climate impact and extreme weather event analyses. Flyingpigeon (Version 1.0)”, ago-2016.

[15] S. Fernández Tejería y A. S. Cofiño, “Despliegue y configuración de un balanceador de carga para servicios THREDDS mediante filosofía DevOps”, jul-2017.

[16] B. N. Lawrence, V. Bennett, J. Churchill, M. Juckes, P. Kershaw, P. Oliver, M. Pritchard, and A. Stephens.” The JASMIN super-data-clúster”. arXiv preprint arXiv:1204.3553, 2012.

[17] “Introduction to the JASMIN Cloud - JASMIN help docs.” [Online]. Disponible: https://help.jasmin.ac.uk/article/282-introduction-to-the-jasmin-cloud

[18] *“Copernicus. La mirada europea sobre la tierra”* (2017). doi 10.2873/2338

[19] *“Annex to the commission Implementing Decision on the adoption of the Work programme 2018 and on the financing of the Copernicus Programme”*. European Commission (Report). Annex 1. Brussels. October 1, 2018. p. 172.

[20] D. Dee, “Copernicus Climate Change Service (C3S)”, p. 37.

[21] B. Raoult, C. Bergeron, A. López Alós, J.-N. Thépaut, y D. Dee, “Climate service develops user-friendly data store”, 2017.

[22] C. Ehbrecht, S. Kindermann, A. Stephens, y S. Denvil, “Web Processing Services for Copernicus Climate Change Service”, p. 1
[23] ECMWF, “C3S Volume II. Global climate projections: data access, product generation and impact of front-line developments. ITT Ref: C3S_34a.” [Online]. Disponible: https://climate.copernicus.eu/sites/default/files/2018-04/C3S_34a_Volume%20II_FINAL.pdf
[24] “Container Orchestration | Kubernetes Management | Rancher.” [Online]. Disponible: https://rancher.com/
[25] “Requirements,” Rancher Labs. [Online]. Disponible: https://rancher.com/docs/rke/latest/en/os/