UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Colas batch en Kubernetes en entornos multitenant y con prioridades

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Cabrejas Peñuelas, Jorge

Tutor/a: Blanquer Espert, Ignacio

CURSO ACADÉMICO: 2023/2024

Resumen

Este trabajo extiende la funcionalidad del componente nativo de Kubernetes, *Job*, implementando una solución ligera de un sistema de colas tradicional utilizando Custom Resource Definitions (CRDs) y Operadores que gestionan recursos personalizados de la gestión de colas. La solución ha demostrado que escala con el número de *pods* hasta un umbral que depende de los recursos que disponga el sistema. Por otra parte, la solución permite reducir los tiempos de ejecución de las tareas.

Palabras clave: Kubernetes, Sistema de colas, Procesamiento por lotes, Custom Resource Definition (CRD).

Abstract

This work extends the functionality of the native Kubernetes Job component, by implementing a lightweight solution for a traditional queuing system using Custom Resource Definitions (CRDs) and Operators that manage custom resources for queue management. The solution has shown scalability with the number of pods up to a threshold that depends on the system's available resources. Additionally, the solution helps reduce the execution time of tasks.

Keywords: Kubernetes, Queue systems, Batch processing, Custom Resource Definition (CRD).

Tabla de contenidos

1 Introducción

1.1 Motivación

Kubernetes se ha convertido en la plataforma de orquestación de contenedores más utilizada en los últimos diez años [1]. Existen proyectos como *Docker Swarm* [2], ya descontinuado, *Apache Mesos* [3], o *Nomad* [4] que adquirieron cierto interés en su momento, pero nada comparado a lo que hoy en día se utiliza Kubernetes. La razón de este éxito frente a sus competidores se debe a su gran capacidad de autoescalado, a su ecosistema de Operadores y a la integración con plataformas en la nube.

Tal es la amplitud del ecosistema de Kubernetes con los principales proveedores en la nube pública que, *Amazon Web Services* (AWS) [5], *Google Cloud* [6] y *Microsoft Azure* [7] ofrecen servicios de Kubernetes totalmente gestionados, sin que los usuarios tengan que gestionar los nodos, las actualizaciones o el mantenimiento.

¿Pero, qué permite hacer Kubernetes? Gracias a Kubernetes los desarrolladores pueden programar y automatizar tareas relacionadas con contenedores a lo largo de todo el ciclo de vida de una aplicación [8]. Por ejemplo, al desplegar una aplicación web en Kubernetes, el sistema puede aumentar el número de réplicas de los *pods* durante un pico de tráfico para evitar interrupciones y reducir ese número cuando se reduzca el tráfico, evitando desperdiciar recursos.

Además, Kubernetes destaca por su diseño modular y altamente extensible. Gracias a su *Application Programming Interface* (API), los desarrolladores pueden ampliar sus funcionalidades mediante la incorporación de nuevos componentes personalizados. Uno de los componentes clave que pueden ser ampliados es el *Job*, que permite ejecutar trabajos o tareas específicas que deben completarse correctamente.

Sin embargo, el componente *Job* de Kubernetes no está pensado para gestionar dinámicamente listas de tareas pendientes como lo haría una cola de trabajos tradicional. Esto significa que no puede agregar nuevas tareas automáticamente a la cola sin intervención manual. Tampoco permite priorizar tareas según criterios específicos, garantizando que aquellos trabajos que sean más prioritarios se ejecuten primero. Además, no se permite establecer dependencias o relaciones de orden entre tareas, lo que es una funcionalidad clave en muchas colas de trabajos. Estas limitaciones hacen que Kubernetes, en su implementación nativa, no sea ideal para gestionar sistemas de colas de trabajos complejas.

En este sentido, el presente Trabajo Fin de Máster (TFM) tiene como objetivo diseñar un sistema ligero de gestión de colas que extienda las funcionalidades nativas de Kubernetes. A continuación, se presentan los objetivos principales de este trabajo.

1.2 Objetivos

El objetivo general de este Trabajo Fin de Máster (TFM) es diseñar, desarrollar y probar un sistema de colas de tareas ligero cuyo *backend* sea el propio clúster de Kubernetes. El sistema debe tener una complejidad operacional reducida, lo que significa que sea fácil de configurar, desplegar y mantener.

Este objetivo general puede ser desarrollado con los siguientes objetivos específicos o acciones concretas a desarrollar:

1) Incorporar mecanismos de priorización de tareas en las colas, permitiendo que aquellas tareas más críticas o urgentes se ejecuten antes que las de menor prioridad.
2) Implementar mecanismos de gestión de colas mediante *Custom Resource Definitions* (CRDs) para permitir la personalización y gestión eficiente de colas y trabajos.
3) Programar la ejecución dinámica de tareas según los recursos disponibles, monitorizando y actualizando el estado de las colas y las tareas que están siendo ejecutadas. Cuando las tareas hayan finalizado, nuevas tareas deben poderse asignar de manera dinámica y autónoma.
4) Manejar el ciclo de vida completo de las tareas en el sistema. La ejecución de una tarea consistirá en la ejecución de un *pod* de Kubernetes. El desarrollo implementado debe monitorizar y limpiar el sistema una vez la tarea haya finalizado.
5) Implementar un sistema de control de concurrencia para limitar el número de trabajos en ejecución simultánea. Esto permite simular el hecho de que los recursos podrían ser limitados, para evitar que el sistema de colas monopolice los recursos del clúster de Kubernetes.
6) Usar comandos estandarizados de la API de Kubernetes para gestionar recursos como *pods*, trabajos y colas, tal que se mantenga la simplicidad operativa.
7) Realizar pruebas que midan las prestaciones del sistema y su escalabilidad con diferentes números de *pods*.

1.3 Metodología

Para cumplir con los objetivos marcados en la sección anterior, se han seguido una serie de pasos para la consecución de los mismos. Esta metodología sugiere una estructura lógica que guía el desarrollo del TFM, desde el análisis de tecnologías existentes, pasando por la implementación técnica hasta la evaluación de resultados.

1. **Estudio de opciones**. Es importante explorar las diferentes soluciones y enfoques ya existentes en la actualidad en relación a ejecutar trabajos en lotes (*batchjobs*). En este sentido, se ha realizado un análisis de las soluciones Kueue [9] y Volcano [10].

2. **Estudio práctico de componentes.** También es importante conocer los componentes específicos Kubernetes que se utilizan para ejecutar trabajos en lotes y cuáles son las deficiencias que hacen plantear nuevas soluciones.
3. **Diseño de la arquitectura.** Una vez ya conocido el estado del arte actual de los sistemas de trabajos en lotes, conviene plantear un arquitectura desplegada sobre Kubernetes que venga a solucionar algunos de los aspectos no abarcados en la actualidad por las soluciones nativas.
4. **Implementación.** En este punto se puede desarrollar un entorno práctico donde se desplieguen y ejecuten trabajos en lotes.
5. **Pruebas.** El siguiente punto es verificar que la implementación funciona correctamente, y evaluar el rendimiento y su escalabilidad.
6. **Conclusiones.** Finalmente, acabamos reflexionando sobre los resultados obtenidos y planteando mejoras o recomendaciones.

La Figura 0 muestra una estimación del coste temporal para la realización del TFM y alcanzar los objetivos marcados. Se estima que el coste total se acerca a los 3 meses.

Figura 0. Estimación del coste temporal.

1.4 Estructura del documento

La estructura del documento está formado por cinco secciones:

1. **Introducción**: esta sección comienza con la motivación del trabajo, marca unos objetivos generales y específicos y establece una metodología lógica del trabajo a realizar.
2. **Estado del arte**: esta sección describe sucintamente la arquitectura de Kubernetes y los proyectos actualmente que se están desarrollando en la comunidad científica internacional de sistemas de colas. Además, plantea los pros y los contras de las diferentes soluciones planteadas.
3. **Diseño de la solución**: describe la arquitectura general de la solución propuesta.
4. **Desarrollo de la solución**: se detalla la implementación de la arquitectura diseñada.
5. **Pruebas**: Se muestran las pruebas que validan el sistema de colas implementado.

1.5 Convenciones

En el código desarrollado en *Python*, se sigue el estilo de código definido en *Flake8*, una herramienta que asegura el cumplimiento de *PEP 8* (estándar oficial de estilo para *Python*), un control de la complejidad, la lectura y el mantenimiento del código, límites de longitud de línea, etc.

A lo largo de toda la memoria, se utilizan los conceptos siguientes con el mismo significado:

- Trabajo, tarea, *Job* y *batchjob*.
- *Slot*, ranura y ejecución de un *pod*.

2 Estado del arte

2.1 Kubernetes

Kubernetes[1] es una plataforma de orquestación de contenedores, originalmente desarrollada por *Google* y ahora mantenida por la *Cloud Native Computing Foundation* (CNCF), que gestiona el despliegue, escalado y operación de aplicaciones en contenedores [1]. Esta gestión permite a los administradores de sistemas desplegar aplicaciones cada vez de forma más sencilla y eficiente sin tener que lidiar manualmente con la infraestructura subyacente.

2.1.1 Alternativas a Kubernetes

Apache Mesos es una plataforma de gestión de clústeres que permite orquestar, no solo contenedores, sino también máquinas virtuales y *frameworks big data* (por ejemplo, *Hadoop* y *Spark*) [3]. Además, se caracteriza por su diseño de alta escalabilidad con el número de nodos y por tener un *scheduler* jerárquico que permite trabajar con diferentes *frameworks* a la vez, tales como *Marathon* para gestionar servicios de larga duración [11], *Chronos* para ejecutar trabajos *batch* y *Spark* para procesamientos de datos distribuidos [12].

Docker Swarm es la solución nativa de *Docker* para orquestar contenedores [2]. Se caracteriza por su facilidad de uso ya que se utilizan los mismos comandos que al usar contenedores *Docker*, su limitada escalabilidad con el número de nodos, por su alta disponibilidad gracias a la replicación de servicios y por la tolerancia a fallos. Esta solución es ideal para entornos pequeños o medianos, donde la curva de aprendizaje es reducida.

Nomad es una solución sencilla de orquestación de la empresa *HashiCorp* que permite no sólo orquestar contenedores, sino también binarios, máquinas virtuales o aplicaciones *Java* [4]. Se caracteriza por su integración con el *stack* tecnológico de *HashiCorp* y por su escalabilidad con el número de nodos para despliegues de tamaño pequeño.

Todas estas soluciones tienen una adopción muy limitada respecto al uso de Kubernetes y únicamente parece que tienen cierto uso cuando se busca una solución simple o compatible con el ecosistema *Docker* o si ya se usan herramientas *HashiCorp*. Por su parte, *Apache Mesos* ha perdido mucha popularidad con el auge de Kubernetes. Por tanto, parece que Kubernetes se ha convertido en el estándar de facto de la orquestación de contenedores. Por este motivo, resultan interesantes soluciones que se construyen a partir de Kubernetes como *Rancher* [13] u *OpenShift* [14].

Rancher es una plataforma de administración y orquestación que facilita la gestión de clústeres de Kubernetes desplegados en la nube o en infraestructura local. Tiene una interfaz gráfica amigable y permite desplegar aplicaciones populares en Kubernetes como

[1] También se conoce Kubernetes como *K8s*, donde la K y la *s* son la letra inicial y final de Kubernetes, respectivamente, y el 8 representa el número de letras entre la K y la s en la palabra Kubernetes.

bases de datos y herramientas de monitoreo, facilitando la integración de los clústeres con entornos de desarrollo y despliegue continuo.

OpenShift es una plataforma de orquestación de contenedores desarrollada por *Red Hat* que es una distribución de Kubernetes modificada. Está orientado a proporcionar una plataforma completa de desarrollo con capacidades *DevOps* e Integración Continua y Despliegue Continuo (CI/CD). Está pensada para grandes empresas con requisitos de seguridad y soporte.

La siguiente subsección presenta la arquitectura general y los componentes principales de Kubernetes.

2.1.2 Arquitectura y componentes

Kubernetes es un gestor de contenedores que organiza las aplicaciones en *pods*, que es la unidad computacional más pequeña desplegada en el sistema [15]. Un *pod* puede contener uno o varios contenedores que comparten almacenamiento, red y especificaciones sobre cómo ejecutar los contenedores. Los *pods* se ejecutan en nodos (trabajadores o en inglés *workers*) que pueden ser máquinas virtuales o máquinas físicas, y son gestionados por un servidor maestro (o varios si se desea incrementar la disponibilidad) que coordina las actividades en el *clúster*[2] de Kubernetes, y que a su vez puede ejecutarse en cualquier nodo del clúster. La Figura 1 muestra la arquitectura general de un clúster de Kubernetes:

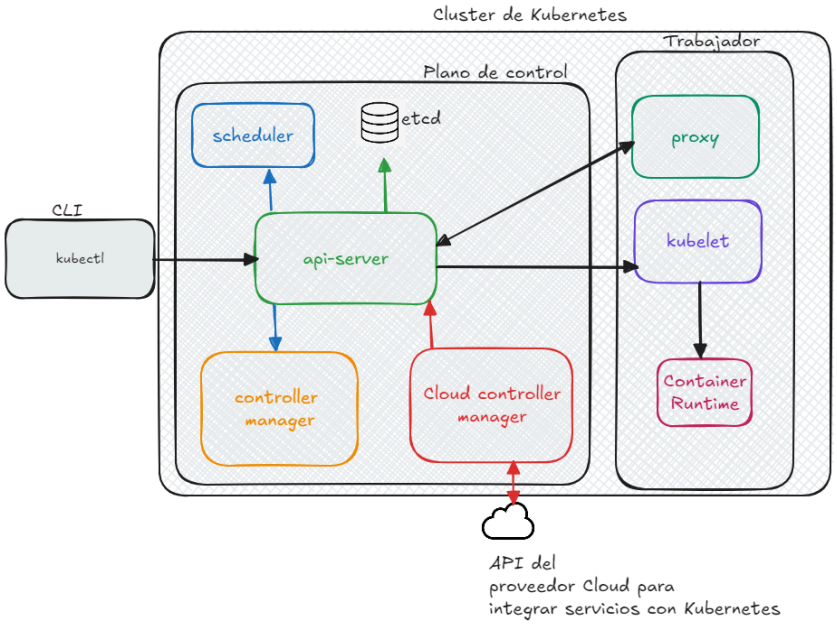

Figura 1. Componentes de la plataforma Kubernetes [1][15].

El servidor maestro corresponde con el plano de control y sus componentes principales son [1][15]:

[2] Clúster se utiliza a menudo en el contexto de Kubernetes para describir el conjunto de máquinas que ejecutan contenedores.

- ***api-server***: es el servidor que recibe las peticiones y actualiza el estado del clúster en la base de datos *etcd*.
- ***etcd***: es un almacén de datos distribuido clave-valor que almacena toda la información del clúster de Kubernetes.
- ***scheduler***: es un componente que está pendiente de los *pods* que no tienen ningún nodo asignado, de modo que selecciona aquel nodo donde poder ejecutar el *pod* que cumpla los requisitos de recursos, restricciones, prioridades y la disponibilidad de recursos.
- ***controller-manager***: es un componente que vigila el estado del clúster, garantizando que este estado esté en conformidad con el estado especificado por los objetos[3] de Kubernetes.
- ***cloud-controller-manager***: es un componente que interactúa con las APIs de los proveedores de servicios en la nube.

Los nodos trabajadores corresponden con el plano de datos y sus componentes principales son [1][15]:

- ***kubelet***: es un agente que se preocupa de que los contenedores estén corriendo en un *pod*. Además, se asegura que los *pods* han sido creados según sus especificaciones.
- ***proxy*:** es un componente que actúa, como su nombre indica, como un *proxy* de red, es decir, es capaz de redirigir el tráfico a los *pods* correctos. Además, es un balanceador de carga para el tráfico dirigido a los servicios y los *pods* en el clúster de Kubernetes.

2.1.3 Extensión de Kubernetes

Una de las ventajas de la arquitectura descrita es que permite añadir nuevas funcionalidades según se requiera. En concreto, dos de las herramientas más poderosas para extender Kubernetes son los *Custom Resource Definitions* (CRDs) y los Operadores.

Los CRDs permiten a los usuarios de Kubernetes definir nuevos objetos personalizados más allá de los objetos que vienen por defecto: *pods*, *deployments* y *services.* Los objetos personalizados actúan como cualquier otro objeto, permitiendo a los usuarios crear, acceder y gestionar sus propias extensiones utilizando la línea de comandos de Kubernetes (*kubectl*) y la API de Kubernetes.

Cuando se crea un CRD, se registra un nuevo tipo de objeto en la API de Kubernetes y el servidor de la API (*api-server*) está listo para manejar este nuevo objeto.

Los Operadores, por su parte, implementan la lógica específica para gestionar los nuevos objetos definidos con un CRD. Utilizan la API de Kubernetes para observar el estado actual del sistema y realizar los ajustes en función del estado deseado definido por el usuario.

[3] Un objeto es una entidad en el sistema de Kubernetes como puede ser un *pod*.

2.2 Soporte de colas en Kubernetes

Kubernetes dispone del objeto *Job* para ejecutar tareas que deben ejecutarse hasta su terminación. A diferencia de los *pods*, que normalmente se diseñan para tareas que no terminan como servidores web, los *Jobs* se crean para ejecutar tareas que deben terminar, como cálculos o procesamientos *batch* que se ejecutan una vez.

A continuación, se muestra un ejemplo de definición de un Job:

```
kind: Job
metadata:
  name: ejemplo-paralelo
spec:
  parallelism: 3
  completions: 3
  backoffLimit: 4
  template:
    spec:
      containers:
      - name: imprimir-mensaje
        image: busybox
        command: ["sh", "-c", "echo Hola Kubernetes!; sleep 10"]
      restartPolicy: OnFailure  # Reintentar si la tarea falla
```

En este ejemplo, el Job está configurado para ejecutar hasta 3 *pods* en paralelo, que deben acabar 3 *pods* exitosamente y que se permiten hasta 4 reintentos en caso de fallo, iniciándose el contenedor.

Aunque los *Jobs* están pensados para procesos por lotes (*batch*), no están diseñados desde su origen para comportarse como un sistema de colas *batch* tradicional donde se administran y se priorizan trabajos en función de políticas y dependencias. El encolado de *Jobs* solo ocurre cuando los recursos del sistema se agotan. Esto significa que no existe una gestión proactiva o un procedimiento que ordene los trabajos en función de la prioridad o políticas específicas hasta que los recursos son insuficientes.

En este sentido, existen proyectos internacionales con una comunidad detrás que están desarrollando diferentes soluciones para la gestión de colas. Por ejemplo, encontramos el proyecto *Kueue* y el proyecto *Volcano.sh* que describimos en las siguientes secciones [9][10].

2.3 Kueue

Kueue es un sistema que funciona sobre Kubernetes diseñado para aplicaciones de procesamiento batch *multitenant*[4] con cuotas. *Kueue* decide cuándo un *Job* debería

[4] Multitenant se refiere a que el sistema soporta múltiples usuarios o inquilinos dentro del mismo clúster que pueden compartir los recursos.

esperar, cuándo debería ser admitido y cuándo debería ser priorizado incluso si existe un *pod* activo.

La Figura 2 muestra la arquitectura general de un sistema *Kueue*.

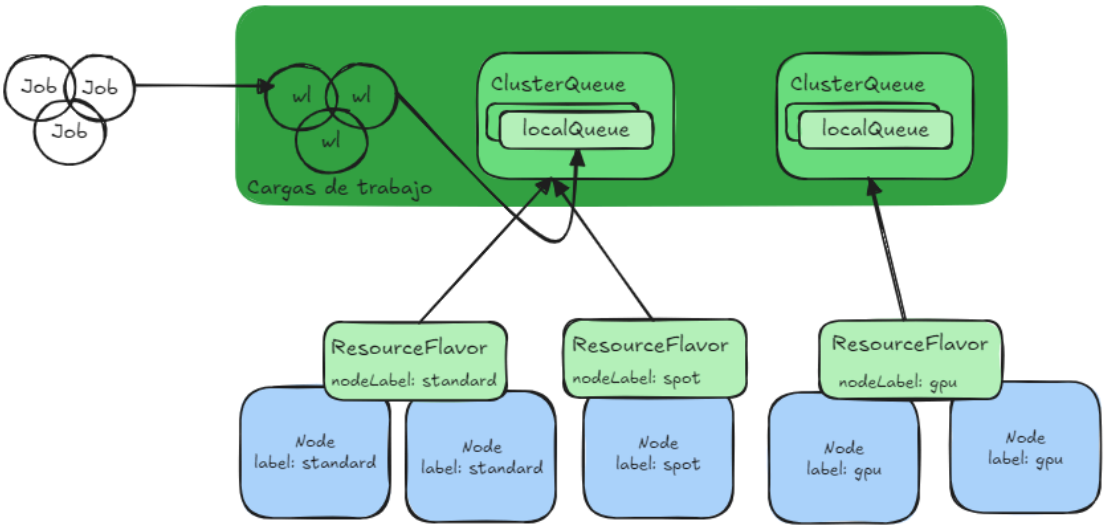

Figura 2. Arquitectura general de un sistema Kueue [16].

Podemos encontrar los siguientes conceptos en el sistema Kueue [9]:

- ***ResourceFlavor***: es un objeto que describe los recursos que están disponibles en un clúster. Típicamente, este objeto se asocia a un grupo de nodos que varían en CPU, GPU o memoria. Por ejemplo, en la figura tenemos nodos que se consideran estándar, nodos *spot* y nodos que contienen una GPU.
- ***ClusterQueue***: es un objeto que define una o más colas en un clúster de Kubernetes que gobiernan *ResourceFlavors*, definen prioridades, cuotas y reglas de reparto de los recursos.
- ***LocalQueue*:** es un objeto que agrupa *Jobs* estrechamente relacionados que pertenecen a un *tenant* dentro de un espacio de nombres.
- ***Workload***: es un *Job* que contiene una necesidad de recursos y prioridades para asegurar que el *Job* se ejecuta exitosamente.
- ***Cohort***: es una agrupación de *ClusterQueues* que pueden tomar prestados recursos no utilizados.

Kueue funciona como se describe en la Figura 3. El procedimiento consiste primero en la creación de los *ResourceFlavors*, *ClusterQueues* y *LocalQueues* por los administradores de la cola de trabajos. Con estos recursos se pueden especificar los recursos disponibles definiendo los límites de uso y las reglas de compartición de recursos. El siguiente paso es crear la aplicación (también conocida como *Job* o *Workload*) que se ejecutará hasta que acabe en la cola definida en *LocalQueue*. Este *Job* será admitido y encolado en la cola basándose en las políticas definidas y en sus requisitos.

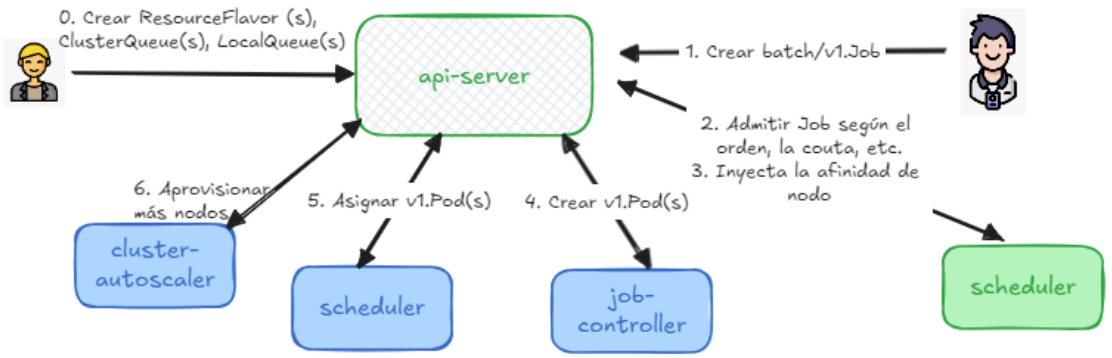

Figura 3. Funcionamiento general de un sistema Kueue [16].

Kueue es un sistema cuya característica principal es que no reemplaza ningún componente de Kubernetes y evita duplicar funcionalidad. El autoescalado, la asignación de los *pods* a los nodos y la gestión del ciclo de vida de un *Job* son responsabilidad del *cluster-autoscaler*, *scheduler* y el *controller-manager*, respectivamente. Nótese que, según la Figura 1, *cluster-autoscaler* no es un componente nativo de Kubernetes, pero se puede instalar y configurar para habilitar el escalado automático de nodos.

2.4 Volcano.sh

El proyecto Volcano es un proyecto desarrollado por la CNCF que proporciona un sistema de colas *batch* avanzado sobre Kubernetes.

La Figura 4 es la arquitectura general del sistema Volcano.

Volcano está formado de los siguientes componentes [10]:

- ***Scheduler*:** es un componente que ofrece funciones más complejas de programación que el *scheduler* nativo de Kubernetes. Permite a *Jobs* que requieren coordinación entre *pods*, políticas basadas en prioridades, desalojar a un *pod* de baja prioridad para que otro *pod* de más alta prioridad se ejecute y gestión de colas.
- ***ControllerManager*:** se encarga de gestionar el ciclo de vida de los CRDs.
- ***Admission*:** es responsable de la validación de los CRDs, es decir, controla las solicitudes de creación o modificación de recursos en el clúster.
- **vcctl:** es el equivalente a *kubectl* en Volcano.

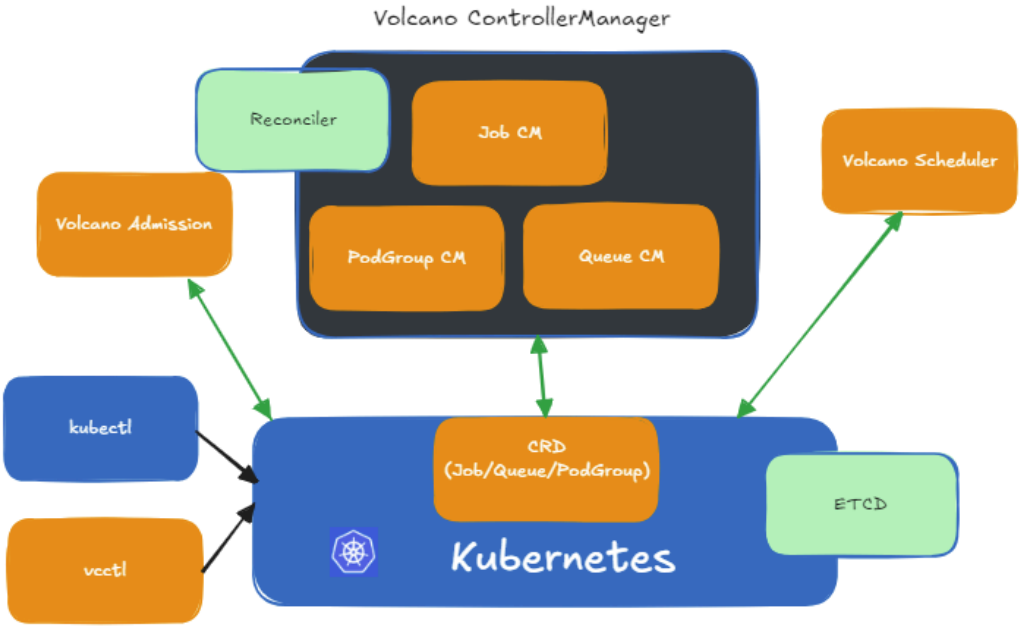

Figura 4. Arquitectura general de Volcano [10].

Volcano define tres tipos de CRDs:

- ***PodGroup*:** es un grupo de *pods* con una fuerte relación y que se usan principalmente en procesamiento *batch*.
- ***Queue***: es una colección de *PodGroups* que adoptan un protocolo *First Input First Output* (FIFO).
- ***VolcanoJob***: es un objeto diferente al *Job* de Kubernetes que proporciona características avanzadas, tales como: *scheduler*, mínimo número de *pods* para que el *Job* sea considerado correctamente ejecutado, la configuración detallada de las tareas a realizar por el *Job*, ciclo de vida, cola y prioridad específica.

2.5 Discusión

Kueue y *Volcano* son dos proyectos diseñados para optimizar la programación de trabajos en clústers de Kubernetes. Ambos surgen por la necesidad de mejorar la ejecución de cargas de trabajo *batch*. Ahora mismo, cuando un usuario crea un *Job*, Kubernetes intenta crear los *pods* e intenta asignarlos a los nodos existentes. Esto puede llevar al sistema al bloqueo ya que el *scheduler* no es capaz de asignarlos a los nodos en momentos de alta demanda. Primero porque no existe un buen control de qué trabajos deberían conseguir recursos ni una política de compartición de recursos. Segundo, una vez que se obtienen los recursos, no existe manera de reasignarlos.

El proyecto *Kueue* es un proyecto de reciente creación, cuya principal característica diferenciadora respecto a otros proyectos como *Volcano*, es que reutiliza los componentes

existentes de Kubernetes como el *scheduler* o el *job-controller*. Esto permite complementar la gestión de colas de una manera más ligera y eficiente.

El proyecto *Volcano* es un proyecto que se construye sobre Kubernetes y que ofrece mecanismos para soportar cargas de trabajo de alto rendimiento en campos como el aprendizaje automático, la bioinformática y en entornos *Big Data*. Fruto de esta madurez, *Volcano* soporta *frameworks* como *Spark*, *TensorFlow*, *PyTorch*, *Flink*, *Argo*, *MindSpore* y *PaddlePaddle*. Sin embargo, este conjunto de mecanismos no se obtiene de manera gratuita, sino que surge gracias a un diseño mucho más ambicioso y complejo que en algunos casos provoca la duplicación de algunas funcionalidades nativas de Kubernetes.

En este sentido, el objetivo de este TFM se alinea con la simplicidad y la complementariedad de *Kueue*.

2.6 Buenas prácticas para la creación de manifiestos YAML

En esta sección, se añaden una serie de buenas prácticas que se recomiendan al crear manifiesto YAML para Kubernetes [17]:

1. Es conveniente utilizar la última versión estable de la API para los componentes que se definan (*v1*), en lugar de versiones Alpha (por ejemplo, *v1alpha1*) o Beta (por ejemplo, *v2beta3*).
2. Utiliza etiquetas (*labels*) y anotaciones (*annotations*) que son en ambos casos, pares clave-valor para organizar y agregar metadatos útiles, respectivamente.
3. No se debería almacenar información sensible como contraseñas en los manifiestos. Se deberían utilizar *Secrets* para gestionar las contraseñas de forma segura.
4. Divide los archivos grandes en varios manifiestos, para facilitar su mantenimiento y comprensión.
5. Intenta generar plantillas reutilizables para despliegues dinámicos con herramientas como Helm o Kustomize [18].

3 Diseño de la solución

3.1 Arquitectura General

En Kubernetes, todo se accede vía API. Para crear *pods*, espacios de nombres (*namespaces*), *deployments*, etc., la API de Kubernetes proporciona *endpoints*. Estos *endpoints* se llaman recursos. Por ejemplo, se puede llamar al recurso */api/v1/pods* para mostrar los objetos *pods* creados.

También es posible crear un recurso propio (*custom resources)* utilizando la API de Kubernetes, como muestra la Figura 6, gracias a los CRDs, que no son más que manifiestos YAML que describen la estructura de un tipo de recurso personalizado. Todos los recursos, ya sean personalizados o no, se almacenan en una base de datos llamada *etcd*.

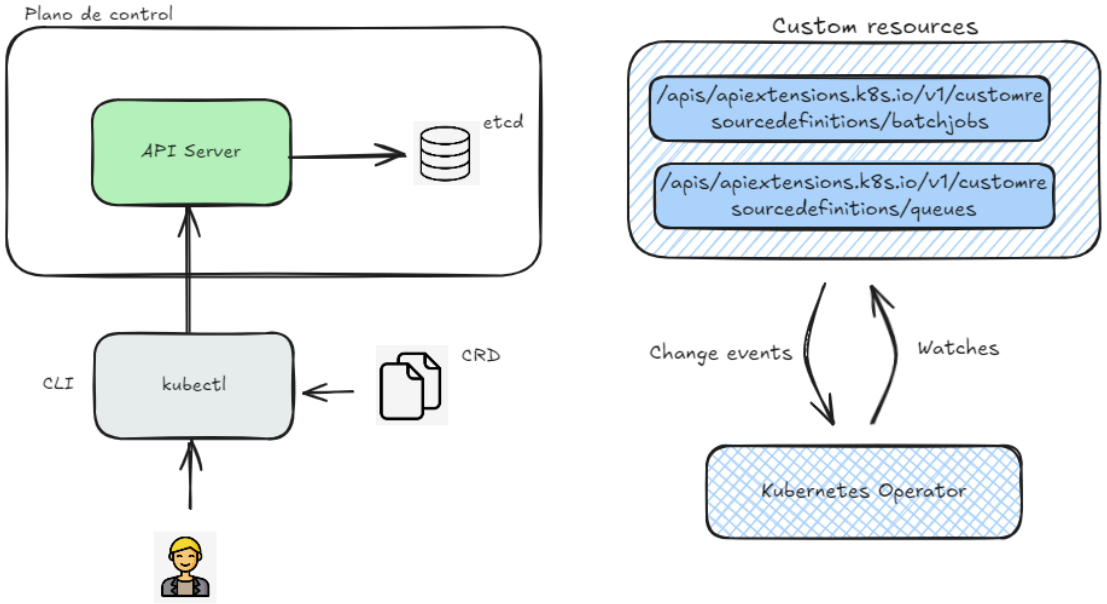

Figura 6. Creación de un recurso personalizado.

Un CRD es un contrato entre el desarrollador y la lógica detrás de ese recurso (Operador). Es este Operador el que, a través de un lenguaje de programación, el que le da sentido a los recursos creados. Por ejemplo, en la Figura 6, en la parte derecha, se observan dos tipos de recursos creados, *batchjobs* y *queues* para representar los trabajos o tareas y las colas. El Operador implementado en este trabajo se basa en *Kubernetes Operator Pythonic Framework* (Kopf) que es una librería para desplegar Operadores en código Python [19].

3.2 Modelo de Colas

Gracias a los CRDs y a los Operadores conseguimos crear un modelo de colas que nos sirva para gestionar trabajos o tareas propias de un sistema de colas tradicional. En la

Figura 7 mostramos el sistema de colas[5] implementado. Por una parte, se observa una cola que almacena diferentes trabajos (*batchjobs*) esperando a ser asignados a recursos (*queuedJob*s) y otros trabajos ya siendo ejecutados (*runningJobs*).

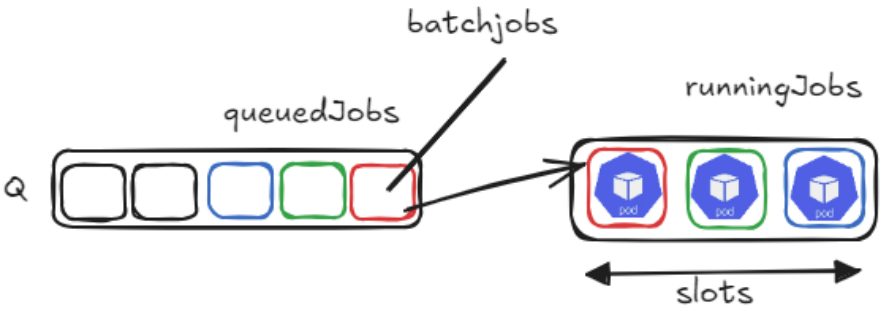

Figura 7. Modelo de colas implementado.

Como se observa, el sistema de colas, definido por un CRD, está formado por una cola de trabajos esperando y por los trabajos que se están ejecutando. Este CRD define el parámetro *slots* que representa el número de tareas que se pueden ejecutar en paralelo. También, se puede ver un *slot* como un *pod*. Por ejemplo, si *slots* es igual a 3 significa que se pueden ejecutar tres trabajos (*batchjobs*) en paralelo. Ese CRD define los trabajos encolados (*queuedJobs)* esperando a ser ejecutados y los trabajos que están siendo ejecutados (*runningJobs)*. El otro CRD es el que representa el trabajo (*batchjob*) que apunta a la cola definida en el otro CRD, una prioridad y detalles relativos a la imagen *Docker* y a un comando a ejecutar. La prioridad permite ejecutar aquellas trabajos más prioritarios antes que aquellos con menor prioridad dentro de la misma cola. A mayor sea el parámetro *priority* mayor será la prioridad[6].

A continuación, se van a definir las funcionalidades principales del Operador desarrollado:

1. ***Scheduling***. Esta funcionalidad consiste en asignar tareas encoladas a recursos del clúster. En concreto, si existe tarea pendiente y existen *slots* disponibles, se procede a crear un *pod* con la imagen y el comando a ejecutar por el mismo. Esta función se ejecuta cada *x* segundos, eligiendo primero los trabajos prioritarios.
2. **Crear sistema de colas**. Se crea una cola con cero trabajos esperando y cero trabajos siendo ejecutados.
3. **Crear trabajo encolado**. Se crea un trabajo dentro de la cola, esperando a que el sistema lo asigne a un recurso.
4. **Borrar trabajo encolado**. Se borran únicamente aquellos trabajos que están esperando a ser asignados a recursos del clúster.

[5] Se define un sistema de colas, y no un cola únicamente, porque el sistema, definido en un CRD, incluye la cola (*queuedJobs*) y los *slots* (*runningJobs*).
[6] También es posible declarar una prioridad a nivel de cola, no solo a nivel de trabajo, para diferenciar aquellas colas más o menos prioritarias. Cabe decir que en la solución planteada se ha implementado el código, pero para simplificar la redacción no se va a exponer.

5. **Monitorizar el estado de los *pods***. Se monitoriza si el *pod* ha terminado satisfactoriamente. Si es así, el *pod* se elimina y el número de slots disponibles se incrementa en uno, listo para ser encolado otro trabajo en la siguiente iteración de *scheduling*. La monitorización se realiza cada *y* segundos.
6. **Priorización de trabajos**. Se ejecutan primero aquellos trabajos cuya prioridad es más alta.
7. **Creación de *pods*.** Una vez que la tarea se va a ejecutar, se desencadena la creación de un *pod* en el clúster que contiene un contenedor *Docker* con una imagen y un comando a ejecutar.
8. **Actualizar el sistema de colas**. Las operaciones previamente descritas implican el cambio de estado del sistema. Esta funcionalidad permite el dinamismo requerido que simula el ciclo de vida de las tareas, desde que se crean hasta que terminan satisfactoriamente.

4 Desarrollo de la solución

4.1 Desarrollo de CRDs y Operadores

En esta sección se describe la implementación del diseño propuesto. El código está subido al repositorio Github cuya URL es https://github.com/jorcabpe/TFM_batchjobs.

Se han definido dos CRDs: *queues* y *batchjobs*. El primero representa el sistema de colas previamente descrito y el segundo los trabajos o tareas a ser encoladas, y finalmente, ejecutadas.

La Figura 8 muestra el código CRD que representa el sistema de colas (*CRD_QueueSystem.yaml*):

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: queues.batch.upv.es
spec:
  group: batch.upv.es
  names:
    plural: queues
    singular: queue
    kind: Queue
    shortNames:
      - que
  scope: Namespaced
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                priority:
                  type: integer
                slots:
                  type: integer
            status:
              type: object
              properties:
                runningJobs:
                  type: array
                  items:
                    type: string
                queuedJobs:
                  type: array
                  items:
                    type: string
```

Figura 8. Código del CRD que define el sistema de colas.

El CRD creado se define dentro de un *namespace*, en la sección *spec.scope*, de tal manera que si se borra el *namespace* todos los objetos dentro del *namespace* se eliminan. El CRD se crea con la siguiente instrucción:

```
kubectl apply -f CRD_QueueSystem.yaml
```

Esto implica que se crea un nuevo *endpoint* en el RESTful API en:

```
/apis/batch.upv.es/v1/namespaces/{namespace}/queues
```

Este *endpoint* se puede usar para crear y gestionar los recursos. La sección *kind* de estos objetos es *Queue*.

Si se quiere borrar el CRD, el servidor desinstalará el *endpoint* y borrará todos los objetos personalizados almacenados:

```
kubectl delete -f CRD_QueueSystem.yaml
```

Las secciones *spec* y *status* tienen funciones muy diferenciadas. La primera sección define los valores utilizados por el usuario cuando se crea o actualiza el recurso. En este caso, *priority* y *slots* representan la configuración deseada para el sistema de colas, como la prioridad que se debe asignar y el número de *slots* o ranuras disponibles. Por otra parte, la sección *status* refleja el estado actual del recurso, el cual se gestiona y actualiza automáticamente por un Operador que hay que definir. En este caso, los campos *runningJobs* y *queuedJobs* representan el estado actual del sistema de colas, mostrando los trabajos que están en ejecución y lo que están en cola, respectivamente.

La Figura 9 representa el código del CRD implementado del *batchJob* (*CRD_BatchJob.yaml*):

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: batchjobs.batch.upv.es
spec:
  group: batch.upv.es
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                queueName:
```

```
                type: string
              priority:
                type: integer
              jobDetails:
                type: object
                properties:
                  image:
                    type: string
                  commands:
                    type: array
                    items:
                      type: string
                  resources:
                    type: object
                    properties:
                      requests:
                        type: object
                        properties:
                          memory:
                            type: string
                          cpu:
                            type: string
                      limits:
                        type: object
                        properties:
                          memory:
                            type: string
                          cpu:
                            type: string
  scope: Namespaced
  names:
    plural: batchjobs
    singular: batchjob
    kind: BatchJob
    shortNames:
    - bj
```

Figura 9. Código del CRD que define los trabajos.

En este caso, únicamente tenemos sección *spec* con los valores *queueName*, *priority* y *jobDetails*. Este último tiene datos de la imagen *Docker*, los comandos a ejecutar dentro del contenedor *Docker* y límites de uso de memoria y CPU del *pod*.

Aquí, de la misma manera que antes, el CRD se crea con el siguiente comando:

```
kubectl apply -f CRD_BatchJob.yaml
```

generando un endpoint en la siguiente ruta:

```
/apis/batch.upv.es/v1/namespaces/{namespace}/batchjobs
```

Si se quiere borrar el CRD, simplemente hay que ejecutar:

```
kubectl delete -f CRD_Batchjob.yaml
```

Una vez que se tienen los CRDs se implementa un Operador en Kubernetes gracias a la librería *kopf* que es una biblioteca popular de *Python* que se caracteriza por su simplicidad y facilidad de uso. Las operaciones de creación, actualización y eliminación de recursos pueden definirse de forma declarativa mediante decoradores *@kopf.on.create*, *@kopf.on.update* y *@kopf.on.delete*, respectivamente.

La Figura 10 muestra la función que implementa un sistema de colas. Esta función creará el recurso al ejecutar el comando *kubectl apply -f*. Es cuando se recibe un evento de que el recurso *Queue* se ha creado, cuando se inicializa y se actualiza el estado.

```python
@kopf.on.create(GROUP, VERSION, QUEUE_PLURAL)
async def create_queue(spec, name, namespace, **kwargs):
    async with event_lock:
        queue_status = {
            'runningJobs': [],
            'queuedJobs': []
        }
        update_queue_status(name, {'status': queue_status})
        logging.info(f"Queue {name} created")
```

Figura 10. Función para crear un sistema de colas.

A las funciones *@kopf.on.create*, *@kopf.on.update* y *@kopf.on.delete*, que funcionan con eventos, se les ha añadido una funcionalidad para adquirir y liberar los recursos asíncronos de manera controlada. Esto evita que una función se ejecute a mitad y que los estados del sistema de colas queden inconsistentes con lo que realmente se está ejecutando.

En la función *update_queue_status* de la Figura 11 se llama a la API de Kubernetes con la función *patch_namespaced_custom_object* (parche parcial) para actualizar los valores de *runningJobs* y *queuedJobs* [20]. Nótese que estos dos valores son los que marcan el estado (*status*) del sistema de colas.

```python
def update_queue_status(queue_name, queue_status):

    try:
        api.patch_namespaced_custom_object(
            group=GROUP, version=VERSION, namespace=NAMESPACE,
            plural=QUEUE_PLURAL, name=queue_name, body=queue_status
        )
    except ApiException as e:
        logging.error(
            f"""Exception when calling
            CustomObjectsApi->patch_namespaced_custom_object: {e}
            """
        )
```

Figura 11. Función para actualizar un recurso personalizado.

Otra de las características a destacar de la librería *kopf* es que tiene un buen tratamiento de errores. Utilizando un bloque *try-except*, si ocurre un error en la API de Kubernetes (como una *ApiException*), el Operador lo captura y lo registra en los logs.

De forma parecida, se ha implementado una función de creación de *batchjobs* en el sistema de las colas (ver Figura 12). Cuando creamos el recurso *batchjob*, la función encola un nuevo *batchjob* dentro del sistema de colas. En particular, almacena el nombre del trabajo dentro de *queuedJobs* y vuelve a llamar a la función *update_queue_status*. Note que, al igual que al crear el sistema de colas, esta función simula la creación de un trabajo, pero realmente se crea el recurso en el sistema de colas al ejecutar el comando *kubectl apply -f*. Es cuando recibe un evento de *batchjob* cuando ejecuta esta función.

```python
@kopf.on.create(GROUP, VERSION, JOB_PLURAL)
async def create_batchjob(spec, name, namespace, **kwargs):
    async with event_lock:
        queue_name = spec.get('queueName')

        if not queue_name:
            logging.error(f"BatchJob {name} does not have a queueName specified.")
            return
        queue = get_custom_object(QUEUE_PLURAL, queue_name)

        if not queue:
            logging.error(
                f"""Queue {queue_name} not found in
                namespace {namespace}."""
            )
            return

        queue_status = queue.get('status', {})
        queue_status['queuedJobs'] = queue_status.get('queuedJobs', []) + [name]
        update_queue_status(queue_name, {'status': queue_status})

        logging.info(f"BatchJob {name} added to queue {queue_name}")
        logging.info(f"queuedJobs: {queue_status['queuedJobs']}")
        logging.info(f"runningJobs: {queue_status['runningJobs']}")
```

Figura 12. Función para crear un *batchjob* en el sistema de colas.

Dado que al crear el *batchjob* con *kubectl apply -f* no aportamos el nombre del sistema de colas, este nombre se obtiene del propio manifiesto del *batchjob* a crear. Con este nombre se puede llamar a la función *get_custom_object* (ver Figura 13) para obtener el recurso gracias a la llamada a la API con la función *get_namespaced_custom_object* [20].

```python
def get_custom_object(object_plural, object_name):
    try:
        return api.get_namespaced_custom_object(
            group=GROUP, version=VERSION, namespace=NAMESPACE,
            plural=object_plural, name=object_name
        )
    except ApiException as e:
        logging.error(
            f"""Exception when calling
            CustomObjectsApi->get_namespaced_custom_object: {e}
            """
        )
        return
```

Figura 13. Función para obtener el recurso.

La Figura 14 muestra la eliminación del sistema de colas llamando a la función *delete_namespaced_custom_object* que elimina cualquier recurso personalizado [20]. Por ejemplo, esta función se llamará cuando se ejecute el comando *kubectl detelete queues --all*.

```python
@kopf.on.delete(GROUP, VERSION, QUEUE_PLURAL)
async def delete_queue(name, namespace, **kwargs):
    async with event_lock:
        try:
            api.delete_namespaced_custom_object(
                group=GROUP,
                version=VERSION,
                namespace=NAMESPACE,
                plural=QUEUE_PLURAL,
                name=name,
                body={},
            )
        except ApiException as e:
            logging.error(
                f"""Exception when calling
                CustomObjectsApi->delete_namespaced_custom_object: {e}
                """
            )
            return

        logging.info(f"Queue {name} removed")
```

Figura 14. Función para eliminar el sistema de colas.

De forma similar, la Figura 15 muestra el código desarrollado para eliminar un *batchjob* dentro del sistema de colas. Para ello, se actualiza el estado del sistema de colas, eliminando el trabajo que se encuentra almacenado en *queuedJobs*. Asumimos que únicamente se pueden eliminar trabajos encolados. Como ejemplo, esta función se llamará cuando se ejecute el comando *kubectl delete batchjobs --all*.

```python
@kopf.on.delete(GROUP, VERSION, JOB_PLURAL)
async def delete_batchjob(spec, name, namespace, **kwargs):
    async with event_lock:
        queue_name = spec.get('queueName')

        queue = get_custom_object(QUEUE_PLURAL, queue_name)

        queue_status = queue.get('status', {})
        queue_status['queuedJobs'] = [
            job
            for job in queue_status.get('queuedJobs', [])
            if job != name and
            job not in queue_status['runningJobs']
        ]

        update_queue_status(queue_name, {'status': queue_status})

        try:
            api.delete_namespaced_custom_object(
                group=GROUP,
                version=VERSION,
                namespace=NAMESPACE,
                plural=JOB_PLURAL,
                name=name,
                body={},
            )
        except ApiException as e:
            logging.error(
                f"""Exception when calling
                CustomObjectsApi->delete_namespaced_custom_object: {e}
                """
            )
            return

        logging.info(f"BatchJob {name} removed from queue {queue_name}")
```

Figura 15. Función para eliminar un *batchjob* del sistema de colas.

Una de las funciones desarrolladas más importantes es el proceso de *scheduling* que asigna trabajos encolados a *slots* o ranuras para ser ejecutados (ver Figura 16). La función ordena los sistemas de colas de más a menos prioridad.

A continuación, para cada sistema de colas, se ordenan los trabajos de más a menos prioridad dentro de la cola gracias a la función *prioritize_batchjobs_of_a_queue* de la Figura 17. Si existen trabajos en la cola y existen ranuras libres, es decir, si el número de trabajos siendo ejecutados es inferior a *slots*, el trabajo se ejecuta. Esto implica la creación de un *pod* y la actualización del sistema de colas a su nuevo estado. Esta funcionalidad se ejecuta cada *x* segundos gracias (en este caso 2 segundos) a *@kopf.timer*, siendo configurable.

La ejecución de un trabajo implica la creación de un *pod* con la función *create_pod* (ver Figura 18). Esta función define unas etiquetas para identificar el *batchjob* y el sistema de

colas. Al crear el *pod* con *V1Pod*, se define el nombre del contenedor, la imagen *Docker*, los comandos y los recursos en memoria y en CPU que tendrá el *pod*.

```python
@kopf.timer(GROUP, VERSION, QUEUE_PLURAL, interval=2)
async def scheduling(spec, namespace, **kwargs):
    async with event_lock:
        list_queues = get_list_queues()

        order_list_queues = sorted(
            list_queues['items'],
            key=lambda k: k['spec'].get('priority', 0), reverse=True
        )

        for queue in order_list_queues:
            queue_status = queue.get('status', {})
            queue_name = queue['metadata']['name']
            prioritize_batchjobs_of_a_queue(queue)
            while (
                (len(queue_status.get('queuedJobs', [])) > 0) and
                (
                    len(queue_status.get('runningJobs', [])) <
                    queue.get('spec', {}).get('slots', 1)
                )
            ):
                job_name = queue_status['queuedJobs'].pop(0)
                queue_status['runningJobs'] += [job_name]
                queue_status['queuedJobs'] = [
                    job
                    for job in queue_status.get('queuedJobs', [])
                    if job != job_name and
                    job not in queue_status['runningJobs']
                ]

                batchjob = get_custom_object(JOB_PLURAL, job_name)
                job_spec = batchjob['spec']['jobDetails']
                image = job_spec['image']
                commands = job_spec['commands']
                resources = job_spec['resources']

                create_pod(job_name, queue_name, image, commands, resources)

                update_queue_status(queue_name, {'status': queue_status})

                logging.info(
                    f"""Starting BatchJob {job_name}
                    from queue {queue_name}"""
                )
                logging.info(f"queuedJobs: {queue_status['queuedJobs']}")
                logging.info(f"runningJobs: {queue_status['runningJobs']}")
```

Figura 16. Función de *scheduling* que gestiona el sistema de colas.

```python
def prioritize_batchjobs_of_a_queue(queue):
    queue_name = queue['metadata']['name']
    try:
        list_batchjobs_in_queues = api.list_namespaced_custom_object(
            group=GROUP, version=VERSION, namespace=NAMESPACE,
            plural=JOB_PLURAL
        )
    except ApiException as e:
        logging.error(
            f"""Exception when calling
            CustomObjectsApi->list_namespaced_custom_object: {e}
            """
        )
    ordered_list_queues = sorted(
        list_batchjobs_in_queues['items'],
        key=lambda k: k['spec'].get('priority', 0), reverse=True
    )

    queue_status = queue.get('status', {})
    queue_status['queuedJobs'] = [
        job['metadata']['name']
        for job in ordered_list_queues
        if job['metadata']['name']
        not in queue_status['runningJobs']
    ]
    update_queue_status(queue_name, {'status': queue_status})
```

Figura 17. Función para ordenar los trabajos de una cola.

```python
def create_pod(job_name, queue_name, image_name, commands, resources):
    labels = {
        "batchjob": job_name,
        "queue": queue_name
    }

    v1 = client.CoreV1Api()
    pod_name = f'pod{job_name}'
    pod = client.V1Pod(
        metadata=client.V1ObjectMeta(name=pod_name, labels=labels),
        spec=client.V1PodSpec(
            containers=[
                client.V1Container(
                    name=job_name,
                    image=image_name,
                    command=["/bin/sh", "-c", " && ".join(commands)],
                    resources=client.V1ResourceRequirements(
                        requests=resources.get('requests', {}),
                        limits=resources.get('limits', {})
                    )
                )
            ],
            restart_policy="Never"
        )
    )

    v1.create_namespaced_pod(namespace=NAMESPACE, body=pod)
```

Figura 18. Función crear un pod.

Para simular el dinamismo de los trabajos que entran a la cola, se ejecutan y se eliminan, se ha desarrollado la función para monitorizar el estado de los *pods* (ver Figura 19). Esta función se ejecuta cada y segundos (en este caso 1 segundo) buscando *pods* que se han ejecutado satisfactoriamente, eliminando tanto el *pod* como el *batchjob* y actualizando el estado del sistema de colas. Las funciones para eliminar el *pod* y el *batchjob* son *delete_namespaced_pod* y *delete_namespaced_custom_object*, respectivamente [20].

```python
@kopf.timer(VERSION, 'pods', interval=1)
async def monitor_pod_status(name, labels, status, **kwargs):
    async with event_lock:
        phase = status.get('phase')

        if phase == 'Succeeded':
            queue_name = labels['queue']
            job_name = labels['batchjob']
            queue = get_custom_object(QUEUE_PLURAL, queue_name)

            queue_status = queue.get('status', {})
            queue_status['runningJobs'] = [
                job
                for job in queue_status.get('runningJobs', [])
                if job != job_name and
                job not in queue_status['queuedJobs']
            ]

            update_queue_status(queue_name, {'status': queue_status})

            v1 = client.CoreV1Api()
            try:
                v1.delete_namespaced_pod(name=name, namespace=NAMESPACE)
            except ApiException as e:
                logging.error(
                    f"""{name}: Exception when calling
                    v1->delete_namespaced_pod: {e}
                    """
                )
                return

            try:
                api.delete_namespaced_custom_object(
                    group=GROUP,
                    version=VERSION,
                    namespace=NAMESPACE,
                    plural=JOB_PLURAL,
                    name=job_name,
                    body={},
                )
            except ApiException as e:
                logging.error(
                    f"""Exception when calling
                    CustomObjectsApi->delete_namespaced_custom_object: {e}
                    """
                )
                return

            logging.info(f"Removing Pod: {name}")
            logging.info(f"Removing Job: {job_name}")
            logging.info(f"queuedJobs: {queue_status['queuedJobs']}")
            logging.info(f"runningJobs: {queue_status['runningJobs']}")
```

Figura 19. Función para monitorizar el estado de los *pods*.

5 Pruebas

5.1 Pruebas de uso

Esta sección muestra todas las pruebas de uso que se han implementado con el fin de verificar que se cumplen los objetivos planteados.

El entorno de pruebas consiste en una máquina virtual creada con *VirtualBo*x con el sistema operativo *Ubuntu 22.04.4 LTS* instalado con 15 GB de RAM y 4 CPUs.

Para simular el entorno de Kubernetes, se utiliza *Minikube v1.32.0*. Este entorno es ideal para desplegar y gestionar aplicaciones de contenedores sin necesidad de un cluster en la nube o en un servidor físico. Además, está diseñado para desarrolladores que desean experimentar en un entorno local y controlado como puede ser este TFM.

Además, usando un solo comando como *minikube start* puedes levantar un clúster básico de Kubernetes en segundos, y de la misma forma, detener el clúster con *minikube stop.*

Para poder verificar tanto el sistema de colas, los CRDs, como el Operador se ha diseñado un conjunto de pasos que intentan mostrar el funcionamiento global del sistema.

Primero, creamos en el sistema los dos CRDs descritos en la sección anterior (ver Figura 20). De esta manera se está extendiendo la API de Kubernetes para permitir definir y gestionar un nuevo tipo de recurso personalizado dentro del clúster. Estos nuevos componentes no vienen definidos de manera nativa en Kubernetes y se ajustan a las necesidades específicas, como la gestión de colas entre otras funcionalidades.

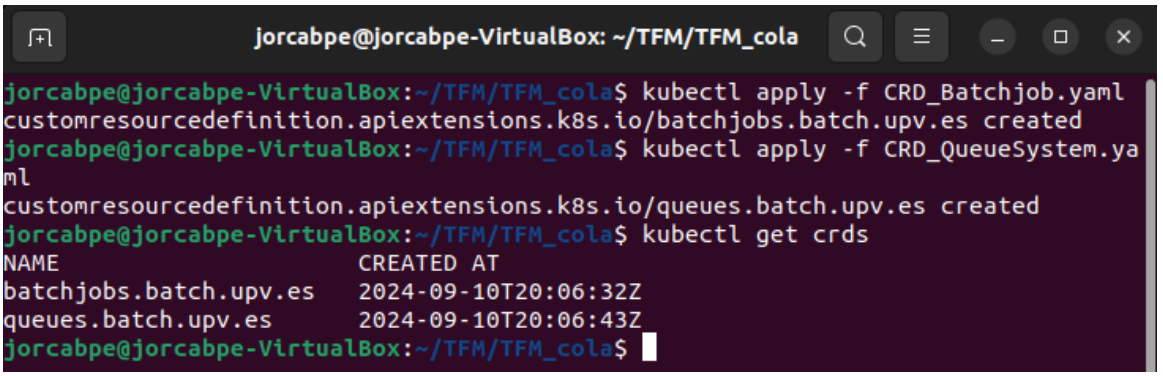

Figura 20. Función que muestra cómo crear los CRDs.

Una vez que el CRD se ha registrado en el clúster, se pueden crear instancias de este nuevo tipo de recurso. Las instancias de este recurso personalizado tendrán la estructura definida en el CRD y se gestionan como cualquier otro recurso nativo de Kubernetes. Si se compara con un lenguaje orientado a objetos, un CRD sería una clase. El concepto de instancia sería similar en ambos lenguajes.

Hasta este momento de la memoria, el Operador implementado no se ha tenido que ejecutar, pero para crear el sistema de colas y para crear trabajos en el sistema se necesita ejecutar. El Operador se queda escuchando, esperando a que lleguen eventos definidos en el propio Operador. Para ello, se ha preparado un entorno virtual que en *Python* se levanta de la siguiente manera:

```
python3 -m venv venv
source venv/bin/activate .
pip install kopf kubernetes
python3 controller_batchjob.py
```

La mejor manera de probar el sistema implementado y comprobar si se cumplen los objetivos generales y específicos, es plantear un experimento práctico. Por ejemplo, se plantea un sistema de colas con 5 trabajos en la cola y 1 pod a usar para ejecutar esos trabajos. Para ello se define la instancia del sistema de colas con prioridad 10 y 1 slot o ranura donde se pueden ejecutar una tarea (ver Figura 21). Note que estos parámetros se han definido en el CRD que define el sistema de colas.

```yaml
apiVersion: batch.upv.es/v1
kind: Queue
metadata:
  name: high-priority-queue
spec:
  priority: 10
  slots: 1
status: {}
```

Figura 21. Manifiesto que representa la instancia del sistema de colas.

Gracias al comando *kubectl apply -f high-priority-queue.yaml*, se ejecuta la función de la Figura 10 con *runningJobs* y *queuedJobs* vacíos. En ese momento, tendríamos la cola vacía.

Ahora podemos crear trabajos (*batchjobs*) que se pueden añadir a la cola. Esto se consigue ejecutando el comando *kubectl apply -f batchjob.yaml*, donde el trabajo se define en la Figura 22.

```yaml
apiVersion: batch.upv.es/v1
kind: BatchJob
metadata:
  name: job1
spec:
  queueName: high-priority-queue
  priority: 5
  jobDetails:
    image: busybox:latest
    commands:
      - "echo Hello, Kubernetes!"
    resources:
      requests:
```

```
        memory: "128Mi"
        cpu: "0.1"
      limits:
        memory: "256Mi"
        cpu: "0.5"
```

Figura 22. Manifiesto que representa la instancia de un trabajo.

Este trabajo (*job1*) y los otros cuatro *batchjobs* (*job2, job3, job4, job5*) pueden tener la misma estructura, pero pueden cambiar, por ejemplo, la prioridad.

La Figura 23 muestra cómo evolucionan los trabajos a lo largo del tiempo (desde arriba a abajo). Primero, se observa que la cola contiene los trabajos *job1*, *job2* y *job3* (paso 1) y que cuando se crea el *job3*, no hay ningún trabajo ejecutándose (paso 2). Después se añade el *job4* (paso 3), y ocurre como antes, que no hay ninguna tarea ejecutándose (paso 4). Después el *job3*, que es el trabajo con más prioridad, empieza a ejecutarse (paso 5). Finalmente, se añade el *job5* a la cola (paso 6). Otra de las cosas importantes a destacar es que únicamente hay una única ranura lista para ejecutar un trabajo. Esto ocurre dado que en el proceso de *scheduling* no entran a ejecutar más de un trabajo a la vez.

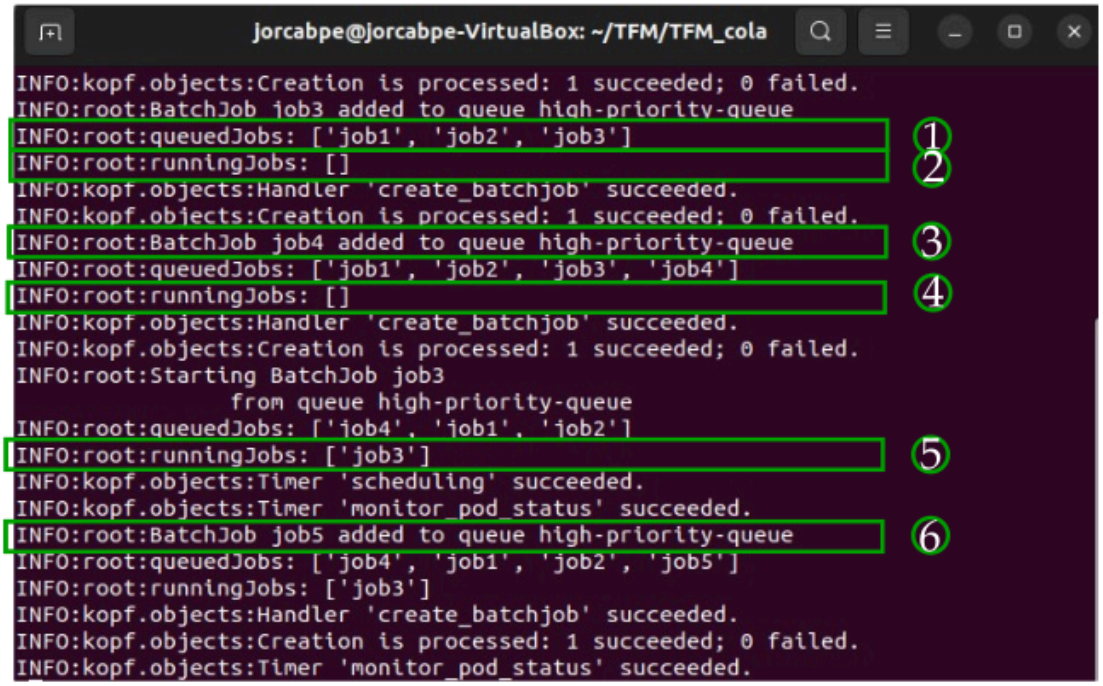

Figura 23. Función que muestra el ciclo de vida de los trabajos en la cola (I).

La Figura 24 muestra la continuación de los trabajos hasta que la cola queda vacía. Note que ha habido una discontinuidad entre el ciclo de vida mostrado en la Figura 23 y lo mostrado en la Figura 24. En este caso, únicamente falta por ejecutar el *job2* (paso 1), que se ejecuta en la siguiente iteración de *scheduling* (paso 2), hasta que no existe ningún trabajo ejecutándose (paso 3) y la cola queda vacía.

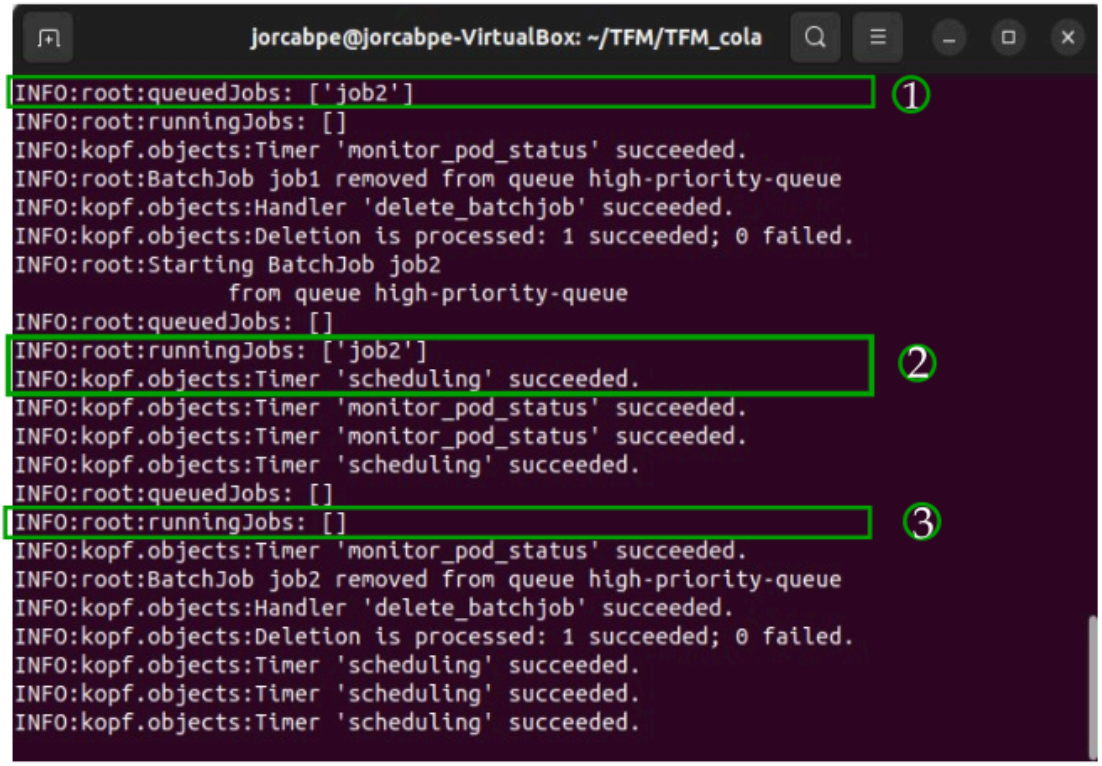

Figura 24. Función que muestra el ciclo de vida de los trabajos en la cola (I).

En este ejemplo, mostrado en ambas figuras, el *scheduler* cada 60 segundos, comprueba si existe algún trabajo listo para ser ejecutado, y si existen recursos libres, ese trabajo se ejecutará en ese momento. Por otra parte, cada 30 segundos se comprobará que la tarea ha terminado, y si es así, eliminará el *pod* y otra tarea podría ser ejecutada cuando el *scheduler* se ejecute. Esta monitorización del estado de los *pods* se realiza cuando se ejecuta la función *monitor_pod_status* (ver Figura 19).

5.2 Pruebas de carga de los servicio y escalado

Analizando los objetivos descritos en la sección 1.2, el único objetivo específico que falta por demostrar es aquel que dice que se deben medir las prestaciones del sistema, así como demostrar su escalabilidad con el número de *pods*. Para hacer pruebas de carga y escalado, utilizamos un paquete de Álgebra Lineal (*LINPACK*) que proporciona rutinas para solucionar sistemas de ecuaciones utilizando principalmente matrices densas [21].

El objetivo de utilizar esta librería no es más que para realizar pruebas con altas exigencias computacionales, no tanto el cálculo en sí mismo. La librería está escrita en el lenguaje de programación C y se ha incluido en un contenedor *Docker* con el *Dockerfile* de la Figura 25:

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y gcc make libc-dev
COPY mtlinpack.c /usr/src/mtlinpack.c
RUN gcc -DSP -O4 /usr/src/mtlinpack.c -lm -DROLL -o /usr/src/mtlinpack.ex2 -w
ENTRYPOINT ["/usr/src/mtlinpack.ex2"]
```

Figura 25. Dockerfile para construir la imagen a descargarse por parte de los trabajos.

Gracias a este *Dockerfile* se ha construido la imagen correspondiente (*docker build -t jorcabpe/mtlinpack-image:latest .*) y se ha subido al repositorio *Dockerhub* (*docker image push jorcabpe/mtlinpack-image*) para que los trabajos, cuando se ejecuten, puedan descargarse la imagen.

En este caso, se ha decidido encolar más trabajos que en el ejemplo anterior. En concreto, se encolan 10 tareas o trabajos, con el manifiesto de la Figura 26:

```
apiVersion: batch.upv.es/v1
kind: BatchJob
metadata:
  name: jobi
spec:
  queueName: high-priority-queue
  priority: XXX
  jobDetails:
    image: jorcabpe/mtlinpack-image:latest
    commands:
      - "/usr/src/mtlinpack.ex2 out.txt YYY"
    resources:
      requests:
        memory: "256Mi"
        cpu: "0.5"
      limits:
        memory: "512Mi"
        cpu: "1"
```

Figura 26. Manifiesto que representa la instancia del trabajo *i-th*.

Note que los valores a cambiar en el manifiesto *batchjob**i**.yaml* son los valores *i*, *XXX* y *YYY*. La tarea *i-th* tendrá la prioridad *XXX(i)* y el parámetro *NTIMES, YYY(i)*. Se definen las prioridades como *XXX(i)=[10, 5, 100, 15, 20, 2, 100, 5, 3, 101]*, y el parámetro *NTIMES*, como *YYY(i) = [1, 2, 5, 3, 6, 1, 1.3, 1.5, 2, 6] * 10000*. El resto de valores del manifiesto no se cambiarán en ningún punto del estudio.

Note además que se han limitado los recursos del trabajo (*pod*), tal que, como mínimo, se solicitan 256 MiB de memoria y medio núcleo de CPU por *pod*, y como máximo, 512 MiB de memoria y un núcleo completo. Es una manera de evitar que el contenedor consuma más de lo necesario.

Los trabajos se lanzan al sistema de colas con una separación de 10 segundos cada uno con el *shell script* de la Figura 27:

```
#!/bin/bash

# Cambiar al directorio donde están los archivos .yaml
cd ./Test

# Bucle para aplicar cada archivo .yaml con un delay de 10 segundos
for i in {1..10}; do
  kubectl apply -f batchjob$i.yaml
  echo "Aplicado batchjob$i.yaml"
  sleep 10
done
```

Figura 27. *Shell script* para lanzar al sistemas de colas 10 trabajos cada 10 segundos.

Las prestaciones que se van a medir es el tiempo transcurrido desde que se ejecuta el anterior *shell script*, hasta que termina el último trabajo y se queda vacía la cola. Para reducir los tiempos de ejecución, se han reducido los tiempos de *scheduling* de 60 a 2 segundos y los tiempos de monitorización de *pods* para su eliminación de 30 a 1 segundo.

La Figura 28 muestra el tiempo transcurrido en segundos con el número de *pods* que se escalan para ejecutar más tareas en paralelo. Por cada uno de los puntos de la figura, se han simulado 3 ejecuciones y se ha realizado la media aritmética de ellas. Los *pods* se han limitado para ser ejecutados entre media y una CPU, y entre 256 MiB y 512 MiB de memoria. Esta limitación se establece para que los *pods* tengan suficientes recursos asignados, pero que no se sobrepasen y que afecten a otros *pods*.

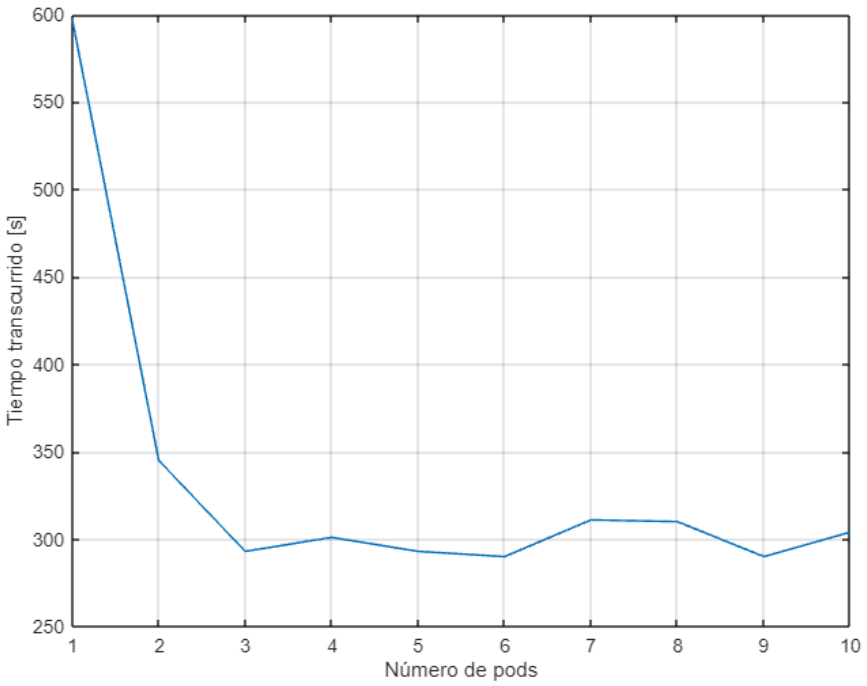

Figura 28. *Prestaciones al ejecutar 10 trabajos variando el número de pods*.

Note que, a medida que aumentamos el número de *pods*, el tiempo transcurrido de ejecución de los 10 trabajos encolados se reduce. Se reduce más de un 50% cuando se usan 3 *pods* frente a un único *pod*. Note además que, más de 3 *pods*, no implica que se mejoren las prestaciones. La explicación de por qué 3 se debe a que la máquina virtual utilizada en las pruebas dispone únicamente de 4 CPUs asignadas. La cuarta CPU se utiliza no para reducir el tiempo transcurrido, sino para tareas propias del sistema operativo. Por tanto, si se incrementaran los recursos de la máquina virtual se mejorarían las prestaciones, reduciendo el tiempo transcurrido. Por tanto, se puede concluir que incrementar el número de pods mejora las prestaciones hasta un cierto umbral que depende del número de CPUs de la máquina virtual.

6 Conclusiones y plan de futuro

En este Trabajo Fin de Máster se ha diseñado, implementado y probado un sistema de colas que viene a extender la funcionalidad básica del componente nativo de Kubernetes, *Job*.

Este sistema se ha diseñado buscando la simplicidad y ligereza, a la vez que obtiene una buena escalabilidad si se incrementan los recursos que liberan la cola de trabajos. En concreto, se ha demostrado que las prestaciones mejoran considerablemente si se incrementa el número de *pods* que ejecutan cargas de trabajo.

Se han alcanzado todos los objetivos planteados, tanto generales como específicos, mostrando mediante pequeños experimentos el funcionamiento básico de la gestión de colas, así como pruebas de carga y de escalado.

El trabajo ha permitido profundizar en diferentes aspectos que se han visto a lo largo del Máster, como son, un orquestador como Kubernetes o contenedores *Docker*. Además, ha permitido aplicar metodologías de investigación, incluyendo técnicas de análisis y obtención de resultados.

Por su parte, el trabajo tiene multitud de líneas de investigación que pueden ser continuadas:

- Una de las características de los trabajos en lotes, es la capacidad de apartar trabajos de baja prioridad que se encuentran actualmente ejecutándose frente a otros que están esperando debido a la falta de recursos. Este mecanismo es uno de los candidatos sin duda a ser implementado.
- Se podrían probar diferentes prioridades a nivel de sistema de colas, ya que es otra de las características de los sistemas avanzados de colas como puede ser *Kueue* (ver Figura 2). Por ejemplo, como ocurre en grandes compañías que disponen de un clúster de nodos, ciertos equipos de trabajo que podrían ser más prioritarios, usarían unos sistemas de colas (Aplicaciones) frente a otros menos (*Marketing*) que usarían otros.
- Se podrían implementar diferentes técnicas de gestión de colas como puede ser *First Input First Output* (FIFO), *Last Input First Output* (*LIFO*) o *Round Robin*.
- En el caso de querer trasladar este trabajo a entornos productivos, las pruebas se deberían realizar con un mayor número de CPUs y memoria. De esta manera, se demostraría que la capacidad de escalado es grande.
- Se podría trabajar con mecanismos (*taints* y *tolerations*) para aislar trabajos en nodos dedicados y evitar que otros trabajos menos prioritarios puedan ejecutarse.

7 Referencias

[1] Kubernetes. https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

[2] Docker. “Classic Swarm: a Docker-native clustering system” https://github.com/docker-archive/classicswarm

[3] Apache Software Foundation. https://mesos.apache.org/

[4] HashiCorp. https://www.nomadproject.io/

[5] Amazon. https://aws.amazon.com/es/kubernetes/

[6] Google. https://cloud.google.com/kubernetes-engine

[7] Azure. https://azure.microsoft.com/es-es/products/kubernetes-service

[8] IBM. https://www.ibm.com/es-es/topics/kubernetes

[9] Kubernetes. https://kueue.sigs.k8s.io/

[10] Cloud Native Computing Foundation. https://volcano.sh/en/

[11] Marathon. https://mesosphere.github.io/marathon/

[12] Chronos. https://mesos.github.io/chronos/

[13] Rancher. https://www.rancher.com/

[14] Red Hat. https://www.redhat.com/en/technologies/cloud-computing/openshift

[15] Nigel Pouton y Pushkar Joglekar. “*The Kubernetes Book*”. Leanpub. 2020.

[16] Kubecon. https://midbai.com/en/post/my-kubecon-china-2023-summary/

[17] Mogenius. https://mogenius.com/blog-posts/best-practices-for-writing-kubernetes-yaml-manifests

[18] Kubernetes. https://kubernetes.io/docs/concepts/workloads/management/

[19] Kopf. https://kopf.readthedocs.io/en/latest/

[20] Kubernetes API client libraries.

https://github.com/kubernetes-client/python/blob/master/kubernetes/docs/CustomObjectsApi.md

[21] Netlib. https://www.netlib.org/benchmark/linpackc

[22] Naciones Unidas.

https://www.un.org/sustainabledevelopment/es/development-agenda/

8 Anexo. Relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS)

En 2015, los estados miembros de las Naciones Unidas, aprobaron 17 Objetivos como parte de la Agenda 2030 para el Desarrollo Sostenible, en la cual se establece un plan para alcanzar los objetivos en 15 años [22]. Estos objetivos se centran en la erradicación de la pobreza, en la protección del planeta y en mejorar las vidas y las perspectivas de las personas. Actualmente, las medidas encaminadas a lograr los Objetivos no avanzan a la velocidad que deberían.

Entre todos los Objetivos de Desarrollo Sostenibles (ODS), este TFM se encuadra principalmente en los siguientes Objetivos (ver Tabla 1):

- ODS 9. **Industria, innovación e infraestructuras:** el presente trabajo se enfoca en el desarrollo de soluciones dentro de las tecnologías avanzadas de la información y la comunicación. Kubernetes es un estándar de facto que permite realizar un desarrollo sostenible y a medida para toda la población interesada.
- ODS 12. **Producción y consumo responsable**: una de las características principales de Kubernetes es que permite maximizar los recursos adaptándose al tráfico experimentado. Por ejemplo, esto permite que las empresas utilicen sus recursos de la mejor manera, obteniendo sus objetivos como empresa, pero a la vez siendo sostenibles medioambientalmente.

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No Procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X

ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**		X		
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Tabla 1. Relación del trabajo con los Objetivos de Desarrollo Sostenible de la Agenda 2030.