UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Gestión de cargas de trabajo batch desde un operador de Kubernetes

Trabajo Fin de Máster

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

AUTOR/A: Galindo Giner, Miguel Ángel

Tutor/a: Blanquer Espert, Ignacio

Cotutor/a: López Huguet, Sergio

CURSO ACADÉMICO: 2021/2022

Gestión de cargas de trabajo batch desde un operador de Kubernetes

Departamento de Sistemas Informáticos y Computación

Máster Universitario en Computación en la Nube y de Altas Prestaciones

Universitat Politècnica de València

Curso Académico 2021-22

Autor

Miguel Ángel Galindo Giner

Directores

Ignacio Blanquer Espert

Sergio López Huguet

ÍNDICE

Resumen Ejecutivo 4

Introducción 5

- Motivación 5
- Objetivos 6
- Plan de desarrollo 6
- Estructura de la memoria 9

Estado del arte 10

- Gestores de Contenedores 10
- Kubernetes 11
- Gestores de Trabajos Batch 15
- Trabajos similares 16

Diseño e Implementación del Entorno 18

- Funcionalidad soportada 18
- Arquitectura de la solución 19
- Componentes de Kubernetes 23
- Despliegue 33

Validación y Pruebas 37

- Entorno de pruebas 37
- Experimentos 37
- Discusión de los resultados 51

Conclusiones 53

Referencias 54

Apéndice 1. Creación imágenes *operador* y Pod 55

Apéndice 2. Relación con los Objetivos de Desarrollo Sostenible (ODS) 56

Resumen Ejecutivo

En el presente Trabajo Fin de Máster, se propone proporcionar a Kubernetes la habilidad de gestionar trabajos batch ejecutados en clústeres SLURM. Para ello se ha diseñado, implementado y validado un *operador* de Kubernetes que realiza la comunicación entre clústers sincronizando los estados en ambos sistemas, facilitando la ejecución de trabajos batch en entornos cloud de forma sencilla.

Los resultados de las validaciones demuestran que se consigue cumplir el objetivo general (proporcionar a K8s la habilidad de gestionar trabajos batch ejecutados en clústeres SLURM) planteado, dotando a K8s la habilidad de gestionar trabajos batch ejecutados en clústeres SLURM.

La aplicabilidad del trabajo final de máster es proporcionar a K8s la posibilidad de ejecución de trabajos tipo batch utilizando otros recursos computacionales y un gestor de trabajos de tipo batch ampliamente utilizado (SLURM), dotando al usuario la capacidad de lanzar/cancelar/monitorizar desde K8s, aunque realmente las ejecuciones se realicen en otro clúster.

Introducción

Kubernetes (K8s)[1] se ha convertido en la plataforma más utilizada para la gestión de aplicaciones multiservicio embebidas en contenedores software. Kubernetes proporciona tolerancia a fallos y elasticidad horizontal, permitiendo a las aplicaciones y a la infraestructura adaptarse a la carga de trabajo. Sin embargo, Kubernetes no está diseñado para gestionar de forma apropiada cargas de trabajo por lotes (trabajos por lotes insertados en una cola de ejecución que se ejecutan de forma desatendida), que suponen una aproximación apropiada para resolver problemas con un alto coste computacional y que pueden abordarse desde un modelo de alta productividad.

En el presente TFM, se pretende proporcionar a K8s la posibilidad de ejecutar trabajos tipo batch utilizando otros recursos computacionales y un gestor de trabajos de tipo batch ampliamente utilizado (SLURM[2]). Para ello, extenderemos K8s mediante un "custom resource definition (CRD)" una forma de extender su API añadiendo funcionalidad, de forma que, el usuario sea capaz de gestionar el ciclo de vida de los trabajos (lanzar/cancelar/monitorizar) desde K8s, aunque realmente las ejecuciones ocurran en otro clúster.

Motivación

Como más adelante explicaremos, K8s no permite de manera nativa la ejecución de trabajos tipo batch, porque está diseñado para gestionar microservicios. Aunque sí incluye el concepto de trabajo[1], no se ajusta al modelo de ejecución de una cola de trabajos batch.

Durante el 2022 ha empezado un proyecto para dotarlo de esta característica, llamado Kueue[3].

El gestor de trabajos SLURM si está diseñado de manera nativa para la ejecución de trabajos tipo batch. Como K8s proporciona una manera no intrusiva de extender su funcionalidad haciendo uso de CRD, podemos utilizar esta característica para poder gestionar trabajos batch desde K8s.

Por ejemplo: Además de tener un K8s en su propia institución, un usuario (ya que usamos SSH) que tiene acceso a infraestructuras de computación basadas en SLURM, puede usar K8s para gestionar los trabajos en todas las infraestructuras.

Este trabajo también pretende permitirme profundizar en el conocimiento de K8s y en concreto en la posibilidad de extender su API usando CRD y experimentar con el concepto de *operador* para poder automatizar tareas.

Objetivos

El objetivo general de este trabajo es proporcionar a K8s la habilidad de gestionar trabajos batch ejecutados en clústeres SLURM.

Para alcanzar este objetivo general se definen los siguientes objetivos específicos:

- Conseguir lanzar trabajos batch desde una máquina remota.
- Definir el ciclo de vida para la gestión de los trabajos desde K8s, (lanzar / cancelar / monitorizar).
- Extender K8s por medio de CRD.
- Capturar los trabajos fallidos notificando el resultado.
- Relanzar trabajos fallidos o relanzar trabajos completados.
- Obtener el resultado de un trabajo desde K8s.
- Implementar el sistema de manera no invasiva con respecto al clúster SLURM conectándose al mismo mediante un usuario regular.
- Realizar la implementación en código abierto.

Plan de desarrollo

Para satisfacer los objetivos descritos en la sección anterior, en primer lugar, se realiza un análisis de las soluciones para posteriormente realizar el diseño de la solución. Con el diseño realizado y teniendo los requerimientos da lugar a la fase de implementación y finalmente se realizarán pruebas para validar la solución.

A continuación, se describen las tareas que se van a realizar en cada objetivo definido:

1. Análisis del estado del arte. (7d)
    - Consultar fuentes bibliográficas sobre Gestores de contenedores, Kubernetes y trabajos similares.
2. Diseño de la solución. (15d)
3. Definición de un Custom Resource Definition de Kubernetes como un nuevo tipo de objeto. (1d)
    - Comenzar por un CRD sencillo para utilizar pequeñas pruebas de concepto.
4. Creación de un *operador* para controlar el ciclo de vida de ese CRD. (7d)
    - Realización de una versión prototipo para la realización de pruebas de concepto con el objetivo de analizar la viabilidad del proyecto
5. Realización de la implementación. (30d)
    - Con el aprendizaje obtenido de las pruebas de concepto, realizar la implementación.

6. Validación mediante una experimentación. (15d)
    - Diseñar un conjunto de experimentos para validar la solución.

Se define el siguiente diagrama de ejecución de las distintas etapas del proyecto final de máster indicando las horas utilizadas en cada etapa:

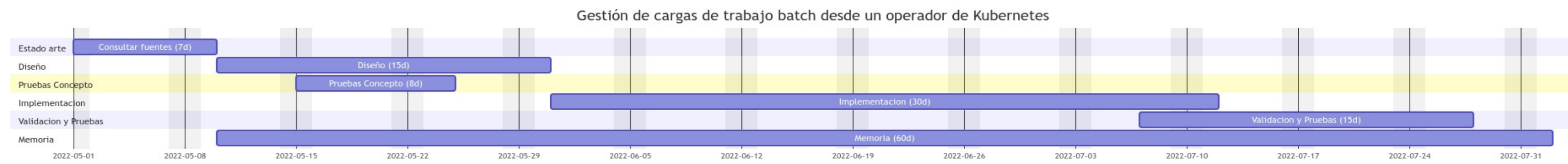

Figura 1: Diagrama Gantt

Estructura de la memoria

La memoria se estructura en cinco secciones principales que se detallan a continuación:

- Introducción: En este punto se desarrolla una breve introducción al proyecto. Se presenta el problema a solucionar, se describen los objetivos a cumplir y se detalla el plan de trabajo.
- Estado del arte: El estado del arte es el estudio de las diferentes tecnologías y soluciones.
- Diseño e Implementación: Se identificarán los requisitos necesarios para realizar el *operador* de Kubernetes y el diseño de la arquitectura. También se detalla la implementación de la arquitectura.
- Validación y Pruebas: Se detallarán las pruebas diseñadas para validar el sistema implementado.
- Conclusiones: Se expondrán las conclusiones obtenidas argumentando si se han cumplido los objetivos y se describen los posibles trabajos futuros.

Estado del arte

A la hora de abordar el trabajo final de máster, en primer lugar, se realiza un análisis del estado actual de las tecnologías y trabajos similares. Por tanto, esta sección comenzará describiendo las tecnologías que hacen posible la ejecución de trabajo batch en entornos cloud y en el ámbito de la supercomputación.

Gestores de Contenedores

Como gestores de contenedores el referente ha sido Docker[3], el cual es una plataforma abierta para el desarrollo, gestión y ejecución de aplicaciones embebidas en contenedores. Docker permite separar aplicaciones de la infraestructura permitiendo una entrega del software de manera eficaz y sencilla. Con Docker se puede gestionar la infraestructura de igual forma que se maneja la aplicación. Docker proporciona la habilidad de empaquetar y ejecutar una aplicación en un entorno llamado contenedor. Los contenedores proporcionan aislamiento, abstrayendo muchos detalles del host y del sistema operativo otorgando en ella seguridad. Los contenedores son ligeros y contienen todo lo necesario para ejecutar las aplicaciones.

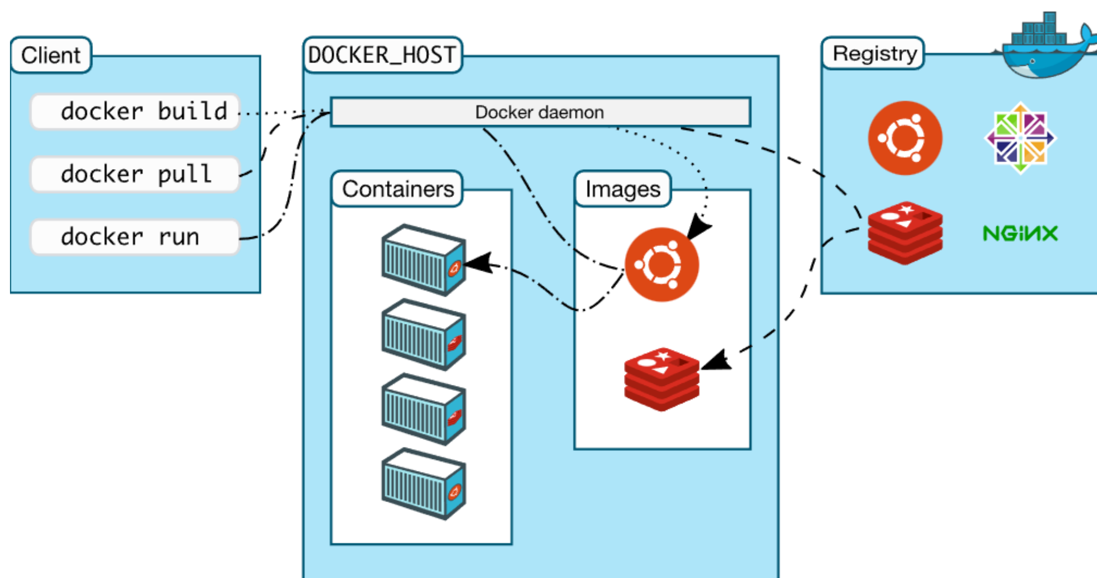

Imagen 2: Imagen arquitectura Docker [3].

Los contenedores empaquetan código y dependencias, siendo una abstracción de la capa de aplicaciones. Múltiples contenedores pueden ejecutarse en el mismo host de forma aislada, pero compartiendo el kernel, lo cual permite un consumo más eficiente de los recursos en comparación con las máquinas virtuales. Esta abstracción hace posible que el entorno de desarrollo sea igual al entorno de producción, lo que reduce inconsistencias mejorando la productividad.

Además del mayoritariamente utilizado Docker existen otros gestores de contenedores como Podman[4], Cri-o[5] o Singularity[6].

Podman es un gestor de contenedores sin demonio, al contrario de lo que sucede con docker que sí utiliza un demonio como controlador. *Podman* está preparado para ejecutar los contenedores sin privilegios root de forma sencilla.

Otro gestor de contenedores es *Cri-o,* está optimizado para Kubernetes y es una alternativa ligera para reemplazar Docker como container runtime.

Por último, Singularity es un gestor de contenedores referente en el ámbito de ejecución de HPC, el cual se caracteriza por proporcionar aislamiento por defecto, se ejecuta bajo el mismo usuario fuera del contenedor como dentro del mismo. No se ganan permisos adicionales dentro del contenedor al contrario de lo que sucede con Docker que por defecto se ejecuta bajo el usuario root.

Kubernetes

Kubernetes (K8s) es una plataforma de orquestación que se puede considerar el estándar de-facto actual. K8s se caracteriza por ser portable y extensible, y a su vez puede orquestar de forma eficiente y robusta las cargas de trabajo y servicios.

Las funciones características de K8s es que permite la automatización de despliegues, el escalado horizontal y contiene herramientas que hacen más sencillo administrar las aplicaciones.

La pieza de cómputo más pequeña que se puede desplegar en K8s son los *Pods*, que están formados por al menos un contenedor. Estos contendores comparten los recursos dentro del *Pod* como puede ser el almacenamiento o la red.

Un clúster de Kubernetes consiste en un conjunto de nodos que ejecutan aplicaciones embebidas en contenedores. Cada clúster al menos debe tener un nodo worker. Los nodos workers albergan los *Pods* (entidad mínima de ejecución) que son los que contienen la carga de trabajo. El "control plane" de K8s gestiona los nodos workers y los recursos. Normalmente en entornos de producción el "control plane" suele estar distribuido entre varias máquinas lo que proporciona tolerancia a fallos.

Un *Pod* se ejecuta por defecto aislado en el clúster, es decir K8s le asigna una *IP* privada aislada del propio host donde se ejecuta. Esta aproximación proporciona aislamiento y seguridad. Por defecto, todos los *Pods* son accesibles dentro del propio clúster de K8s.

Kubernetes proporciona un mecanismo para definir el tipo de comunicación con el exterior que necesita un determinado *Pod*. Este mecanismo tiene el nombre de Servicio y se implementa en K8s mediante diferentes modelos (*ClusterIP*, *NodePort* y *LoadBalancer*).

Los objetos en K8s están creados de forma declarativa haciendo uso de ficheros *YAML*[7] (YAML Ain't Markup Language). *YAML* es un leguaje de serialización de datos que es soportado por bibliotecas software de una gran cantidad de lenguajes de programación.

Por ejemplo, el siguiente bloque de código muestra el código *YAML* para crear un *Pod* que ejecuta un servidor web ngnix:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:stable-alpine
        ports:
        - containerPort: 80
```

Figura 2: Ejemplo código YAML.

Componentes del "control plane"

El control plane se encarga de tomar decisiones sobre todo el clúster (como la planificación o scheduling) detectando y respondiendo a los eventos que se producen en el clúster.

Los componentes que forman el "control plane" pueden ejecutarse en cualquier máquina, sin embargo, por simplicidad suelen ejecutarse en la misma máquina, donde no se suele permitir ejecutar *Pods* del usuario. Los componentes son:

- **Kube-apiserver**: El "apiserver" es un componente que expone el API de Kubernetes.

  El kube-apiserver está diseñado para ser escalable, si se despliegan más instancias se equilibrará el tráfico ente ellas.

  Permite a los usuarios comunicarse con diferentes partes del clúster y componentes externos comunicarse entre ellos.

  La API de Kubernetes permite consultar y manipular el estado de los objetos API por ejemplo (*Pods*, *Namespaces*, *ConfigMaps*, y *Events*).

- **Etcd:** El componente *etcd* es una base de datos clave-valor consistente y de alta disponibilidad para almacenar los datos de todo el clúster.
- **Kube-scheluder:** Componente del control plane que gestiona los nuevos *Pods* que no tienen ningún nodo asignado, asignándole uno para su ejecución.

  El *scheluder* tiene diferentes factores para tener en cuenta a la hora de asignar un nodo a un *Pod*, factores como requerimientos de recursos, restricciones en las políticas/hardware/software.

- **Kube-controller-manager:** El componente "controller-manager" es un proceso que ejecuta los controladores.

Cada controlador es un proceso separado pero que para reducir su complejidad esta todo compilado en un solo binario y se ejecuta en un solo proceso.

Algunos tipos de controladores son, controlador de los nodos, controlador de los Jobs o controlador Endpoints.

Componentes de los Nodos

Los componentes de los nodos, proporcionan el entorno de ejecución de Kubernetes y mantienen en ejecución los *Pods*.

- **Kubelet:** Es un agente que se encuentra en ejecución en cada nodo del clúster, se encarga de que los contenedores se ejecuten en los *Pods*. Kubelet no se encarga de contenedores que no ha creado Kubernetes.
- **Kube-proxy:** El componente kube-proxy es un proxy de la red que se ejecuta en cada nodo, implementando parte del concepto de *Servicio* de Kubernetes. Se encarga de mantener las reglas de red en cada nodo. Kube-proxy usa la capa de filtrado de paquetes de red del sistema operativo si existe una y si está habilitada.
- **Entorno ejecución containers:** Este componente es el encargado de ejecutar los contenedores en cada nodo.

  Kubernetes soporta múltiples entornos de ejecución de contenedores como containerd, CRI-O o cualquier otra implementación de Kubernetes CRI (Container Runtime Interface).

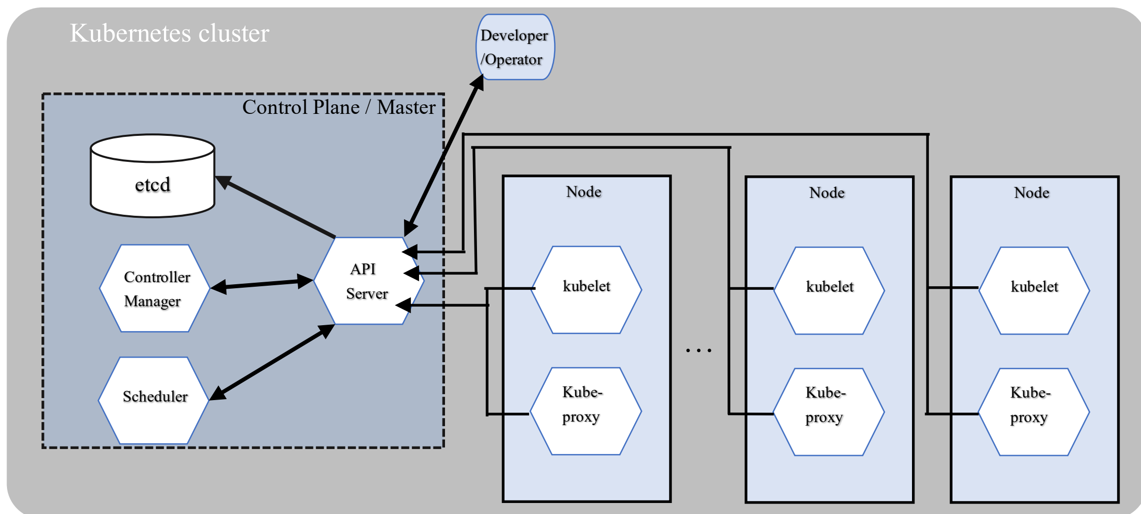

Imagen 3. Arquitectura Kubernetes

Un concepto muy importate en K8s son los *namespaces, los cuales* que permiten aislar un conjunto de recursos dentro de un clúster.

Si bien los *Pods* son la unidad lógica más pequeña de ejecución, K8s proporciona los objetos *Deployment, los cuales* permiten definir de forma declarativa el estado deseado de uno o varios pods Los controladores de K8s se encargan de asegurarse que los pods se encuentren en el estado deseado.

Un apartado importante es la seguridad. K8s utiliza diferentes técnicas y utiliza *RBAC* (*Role Based Access Control*) para asegurar que los usuarios y la carga de trabajo tienen los permisos adecuados para ejecutar su rol. Existen roles a nivel de *namespace* o a nivel de clúster.

Con los objetos *rolebinding* y clusterolebinding se relacionan, respectivamente, los roles definidos con los usuarios a los que se les asigna a nivel de clúster o a nivel de namespace

Por último, K8s proporciona una manera de extender su API que permite empaquetar código para gestionar aplicaciones, tareas repetitivas o de dotar de nuevos objetos con su funcionalidad. Se trata del concepto de *Operador* que es un controlador específico de aplicaciones que amplía las funciones de la API de Kubernetes sin modificarla, pudiendo crear, configurar y gestionar aplicaciones en nombre de un usuario. Los operadores siguen el principio llamado control-loop donde se controlan el estado del objeto contra el estado definido.

Un *operador* juega el rol de automatizar todas las tareas que realizan los operadores humanos sobre una aplicación o servicio.

Un ejemplo de cómo se crean objetos en K8s de forma declarativa, es utilizado *YAML*:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:stable-alpine
        ports:
        - containerPort: 80
```

En cuadro anterior se observa la especificación *YAML* la cual define la imagen a utilizar en este caso *nginx:stable-alpine,* las réplicas, es decir el número de instancias que se crean, y dentro del containers se especifica el puerto abierto hacia el clúster y de esta manera se podría utilizar un objeto *Service* para hacer accesible el *Pod* al exterior utilizando cualquier modelo de los expuesto anteriormente.

La instanciación del *Pod* en K8s se realiza mediante el comando `apply`:

```
kubectl apply -f ejemploPod.yaml
```

Verificando su correcta creación con el comando `get`:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob$ kubectl get pod
NAME                     READY   STATUS    RESTARTS   AGE
nginx-7fc844f74-ff7hx    1/1     Running   0          65s
```

Imagen 4. Obtención de los pods del namespace por defecto (default).

Para ver los logs del *Pod*:

```
kubectl logs nginx-7fc844f74-ff7hx
```

```
migagi@ubuntu:~/slumJob_gitlab/slumjob$ kubectl logs nginx-7fc844f74-ff7hx
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
2022/08/06 19:44:01 [notice] 1#1: using the "epoll" event method
2022/08/06 19:44:01 [notice] 1#1: nginx/1.22.0
2022/08/06 19:44:01 [notice] 1#1: built by gcc 11.2.1 20220219 (Alpine 11.2.1_git20220219)
2022/08/06 19:44:01 [notice] 1#1: OS: Linux 4.15.0-124-generic
2022/08/06 19:44:01 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2022/08/06 19:44:01 [notice] 1#1: start worker processes
2022/08/06 19:44:01 [notice] 1#1: start worker process 34
2022/08/06 19:44:01 [notice] 1#1: start worker process 35
2022/08/06 19:44:01 [notice] 1#1: start worker process 36
2022/08/06 19:44:01 [notice] 1#1: start worker process 37
```

Imagen 5. Obtención de los logs del Pod.

Gestores de Trabajos Batch

Una cola de trabajo es una lista de trabajos pendientes de asignar recursos para su realización. Un trabajo por lotes o batch job es la unidad básica de ejecución al cual se le asignan recursos para su ejecución por un determinado tiempo. Los usuarios crean scripts para enviar sus trabajos, especificando los recursos necesarios como el tiempo de ejecución, procesamiento o memoria y es trabajo del gestor de recursos atender las solicitudes de acuerdo con el número de trabajos pendientes –en cola- y a la prioridad del trabajo.

Los sistemas de colas más utilizados son SLURM[2] y Torque[8].

SLURM (Simple Linux Utility for Resource Management), es un proyecto open source se creó para cubrir el hueco que había para la gestión de clúster open source, diseñado para ser flexible y tolerante a errores, puede ser portado fácilmente a otros clústers de diferente tamaño y arquitectura.

La principal aportación es crear una herramienta que cualquiera puede utilizar para gestionar de forma eficiente clústeres de diferentes tamaños y arquitectura.

TORQUE (Terascale Open-source Resource and Queue manger) es otro gestor de recursos también es open source basado en el original PBS, sus principales características escalabilidad, tolerancia a fallos, usabilidad y funcionalidad.

Trabajos similares

En Kubernetes existe el concepto de *job*, aunque su orientación es diferente. Un *Job* de Kubernetes puede crear uno o más *Pods* y controla el número de ellos que terminan correctamente. Cuando se alcanza el número de ejecuciones satisfactorias indicado, el *Job* finaliza correctamente. No existe el concepto de cola de ejecución, por lo que diferentes trabajos competirían por los recursos. Sería posible desplegar de forma concurrente una gran cantidad de trabajos limitados por recursos y dejar que K8s los gestione (suspendiendo aquellos para los que no hay recursos suficientes). Esto requeriría crear un espacio separado y limitar los recursos en el espacio para evitar efectos colaterales con otras cargas de proceso.

Como limitación, el objeto *Job* no se diseñó para dar soporte a procesos paralelos estrechamente comunicados, como los que comúnmente se encuentran en la computación científica, para ciertos tipos de procesos batch que necesitan de redes de alta eficiencia en las que K8s más concretamente la virtualización de la capa de red no es tan eficiente como la real y de trabajos que requieran del uso extensivo del paralelismo como los trabajos en MPI (Message Passing Interface) para estos casos y para aprovechar toda la infraestructura se ve la necesidad de desarrollar un conector en la que permite unir ambos mundos.

Existen una serie de trabajos en los que se han desarrollado conectores Kubernetes-HPC HPC-Connector[9]. En este trabajo se presenta la necesidad de crear un conector precisamente para ofrecer la unión de ambas infraestructuras, permite integrar cargas de trabajo en la infraestructura cloud sin necesidad de tener privilegios, proporcionando a los usuarios la misma interfaz para diferentes HPC. Nos hemos basado en este desarrollo para el presente trabajo final de máster.

Una herramienta que nació para resolver las limitaciones indicadas en Kubernetes es Volcano[10], una herramienta diseñada para implementar colas y el uso de MPI en Kubernetes. El objetivo de esta herramienta es proporcionar a los clústers K8s de las características necesarias para poder ejecutar trabajos batch. Sin embargo, el objetivo de este trabajo es poder conectarse a clústers ya existentes que ya estén siendo utilizados para el procesamiento de altas prestaciones.

Con estas limitaciones surge la necesidad de desarrollar un componente sencillo dentro del API de Kubernetes que permita la fusión de ambos “mundos” ya que los trabajos mencionados tienen la capacidad de ejecutarse desde Kubernetes, pero no están integrados. Lo que se propone en el presente trabajo final de máster es crear un componente integrado en Kubernetes que permita lanzar trabajos en un clúster HTC

permitiendo manejar todo el ciclo de vida como si fuera un componente nativo. Como se mencionó anteriormente, se ha utilizado la idea del trabajo HPC-CONECTOR que permite reutilizar los trabajos ya implementados para HPC desde Kubernetes o crear nuevos sin necesidad de tener que aprender un API nueva o tener que desplegar los recursos computacionales en el propio clúster, como pasaría en el caso de VOLCANO.

Diseño e Implementación del Entorno

En la presente sección se va a explicar el diseño de la solución propuesta y la funcionalidad implementada, describiendo la arquitectura de ambos sistemas (Kubernetes y SLURM) y su integración mediante el *operador* de Kubernetes implementado para la fusión de ambas arquitecturas.

El objetivo ha sido desarrollar un componente para quien esté acostumbrado a trabajar con objetos K8s no le fuera difícil familiarizarse con este nuevo objeto, pero a la vez le fuera sencillo reconocer los estados manejados en SLURM.

Funcionalidad soportada

En esta sección se describirá el ciclo de vida de un trabajo batch y las operaciones que va a ofrecer definiendo la funcionalidad soportada.

El ciclo de vida de un trabajo batch manejado por SLURM establece cinco estados (PENDING, RUNNING, CANCELED, FAILED o COMPLETED). Se ha optado por definir estos mismos estados en el objeto creado en Kubernetes de forma que los usuarios que ya están familiarizados con el significado de cada estado les resulte más sencillo.

La funcionalidad soportada es prácticamente la misma que se puede realizar al ejecutar trabajos en colas SLURM:

1. Ejecución (submission): Se podrán ejecutar trabajos desde K8s.
2. Cancelación: Se va a tener en cuenta la cancelación del trabajo por parte del usuario tanto si se cancela desde K8s como si se cancela desde el clúster SLURM.
3. Obtención del estado de los trabajos: Se permitirá monitorizar, listar los trabajos que haya creado el usuario.

Se pueden crear trabajos batch definiendo en el propio objeto el código a ejecutar en el clúster.

Para la modificación de trabajos en K8s se limitará a cuando el objeto K8s este en unos determinados estados (FAILED, ERROR, CANCELED o COMPLETED), y se podrá modificar el trabajo, volviéndose a ejecutar con el cambio realizado. Este funcionamiento le da flexibilidad, por ejemplo, con la corrección de errores o en la reutilización del código en trabajos similares, no haciendo falta crear un trabajo de cero.

Cuando un objeto se encuentra en ejecución en K8s y es borrado, el *operador* se comunica con el clúster SLURM para cancelar el trabajo (invocando en el clúster el comando SCANCEL) antes de ser borrado en Kubernetes.

Por otro lado, si el trabajo batch está ejecutándose en el clúster SLURM y es cancelado (por ejemplo por el usuario) o ha ocurrido un error, el *operador* debe detectar esta situación y reflejarla en el estado del objeto K8s.

El *operador* maneja de forma correcta la ejecución de N objetos simultáneos, actualizando el estado en el objeto correspondiente.

Una vez el trabajo ha finalizado, el propio *operador* crea un *Pod* con el mismo nombre que el proporcionado al objeto *slurmJob*, donde el *Pod* recupera el output del trabajo en el clúster SLURM y lo vuelca en los logs, permitiendo visualizar los logs del trabajo lanzado en el clúster SLURM desde Kubernetes de forma nativa.

A continuación, se muestra un diagrama de estado con el flujo soportado por el sistema.

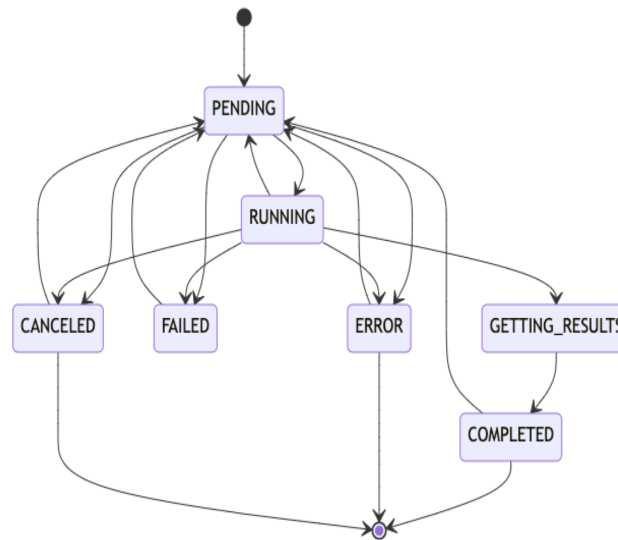

Imagen 6. Diagrama de estado

Arquitectura de la solución

Para satisfacer los requisitos establecidos, planteamos una arquitectura dividida en dos partes: Parte cloud (un clúster K8s) y parte HPC (un clúster SLURM).

Para la parte del clúster HPC, la arquitectura debe permitir utilizar el clúster de forma no invasiva por lo que no se tiene que hacer ningún desarrollo en el lado HPC. Para la ejecución de los trabajos, el *operador* crea el script usado para definir un trabajo SLURM en K8s y lo envía para su ejecución al clúster SLURM, para lo que el *operador* lo deja en carpeta personal del usuario indicado en un objeto K8s *Secret*, utilizado por un usuario regular.

En la parte cloud es donde reside la lógica. Para poder extender el clúster de K8s con la nueva funcionalidad, vamos a utilizar el concepto de *operador* que permite añadir funcionalidad en K8s sin modificar el propio K8s. Para explicar el diseño diferenciamos los siguientes puntos:

1. Definición objeto *CRD* ***slurmjob.***
2. Creación del objeto.
3. Modificación del objeto.

4. Borrado del objeto.
5. Monitorización del estado del objeto.
6. Finalización del trabajo.

1. Definición objeto *CRD* ***slurmjob:***

Para poder crear un *operador*, se necesita definir el nuevo tipo objeto kubernetes al que el *operador* gestiona, al que llamaremos ***slurmjob***. El nuevo objeto ***slurmjob*** almacena la misma información de un trabajo que se obtiene del clúster SLURM (jobid, partition, name, user, state y time), además de todos los datos necesarios para poder conectar con el clúster SLURM utilizando SSH. Para crear este nuevo objeto se crea un *CRD* llamado ***slurmjob*** donde se define los datos anteriormente mencionados, de los cuales definimos que todos son obligatorios, para que el propio K8s los valide antes de crear el objeto.

El objeto tipo ***slurmjob*** en una columna especifica guarda la definición del trabajo (script), es decir donde se indica el código del trabajo batch a ejecutar, para las credenciales de acceso al clúster SLURM, se guardan en un objeto de diferente tipo *Secrets* y en el objeto ***slurmjob*** solo se indica el nombre del objeto *Secret*.

2. Creación del objeto por parte del usuario.

En K8s, cuando el usuario crea el fichero *YAML* con la definición del objeto ***slurmjob***, éste indica el código a ejecutar, las credenciales, el host y el puerto. K8s valida que estén definidos todos los datos, y es el *operador* el que se encarga de validar además que el objeto *Secret* exista, y que contenga las credenciales informadas. Con la información guardada en la columna del código, crea el fichero script que envía al clúster SLURM. Posteriormente el *operador* envía al clúster SLURM el comando `SBATCH` (comando SLURM que envía el trabajo a la cola de ejecución).

3. Modificación del objeto slurmjob por parte del *operador*:

Se permite modificar un objeto que se encuentre en los estados COMPLETED, FAILED o CANCELED.

Cuando se modifica un trabajo, al igual que en la creación se vuelve a validar que estén definidos todos los datos, se crea el nuevo fichero y se vuelve a enviar.

4. Borrado del objeto slurmjob:

Al eliminar el objeto slurmJob siendo su estado RUNNING (en ejecución), el *operador* se conecta al clúster SLURM cancelando el trabajo (comando `SCANCEL`) antes de eliminar el objeto en K8s y sus objetos dependientes (*Pod*). Una vez borrado el trabajo en el clúster SLURM a continuación, se borra el objeto en K8s.

En los demás estados simplemente se eliminar el objeto K8s.

5. Monitorización del estado del objeto:

El *operador* es responsable de regular el estado del trabajo. Para ello, monitoriza el estado de la ejecución en el clúster SLURM y lo usa para actualizar el estado del objeto slurmjob en K8s, consiguiedo así un reflejo del estado que tiene en el clúster SLURM. Por ejemplo, en el caso de que por cualquier motivo se cancele un trabajo en ejecución en el clúster SLURM, el *operador* al recuperar el estado del trabajo obtiene que su estado es CANCELED (cancelado) y actualizara el estado en el lado K8s.

6. Finalización del trabajo y recuperación del output:

Cuando el trabajo batch finaliza, el *operador* crea un *Pod* que se encarga de recuperar del clúster SLURM el fichero con el resultado del trabajo batch y lo imprime por la salida estándar (para que se guarde en los logs del *Pod)*. El *Pod* tiene el mismo nombre que el definido para el objeto *slurmJob*.

Por último, un punto siempre delicado al trabajar con K8s es la gestión de las credenciales, es complicado de solucionar, finalmente se opta que el usuario tiene que crear un objeto Secret con su usuario y contraseña, esta solución no es ideal desde el punto de vista de la seguridad, ya que la contraseña está codificada en base64

Por último, se muestra un diagrama con la arquitectura planteada.

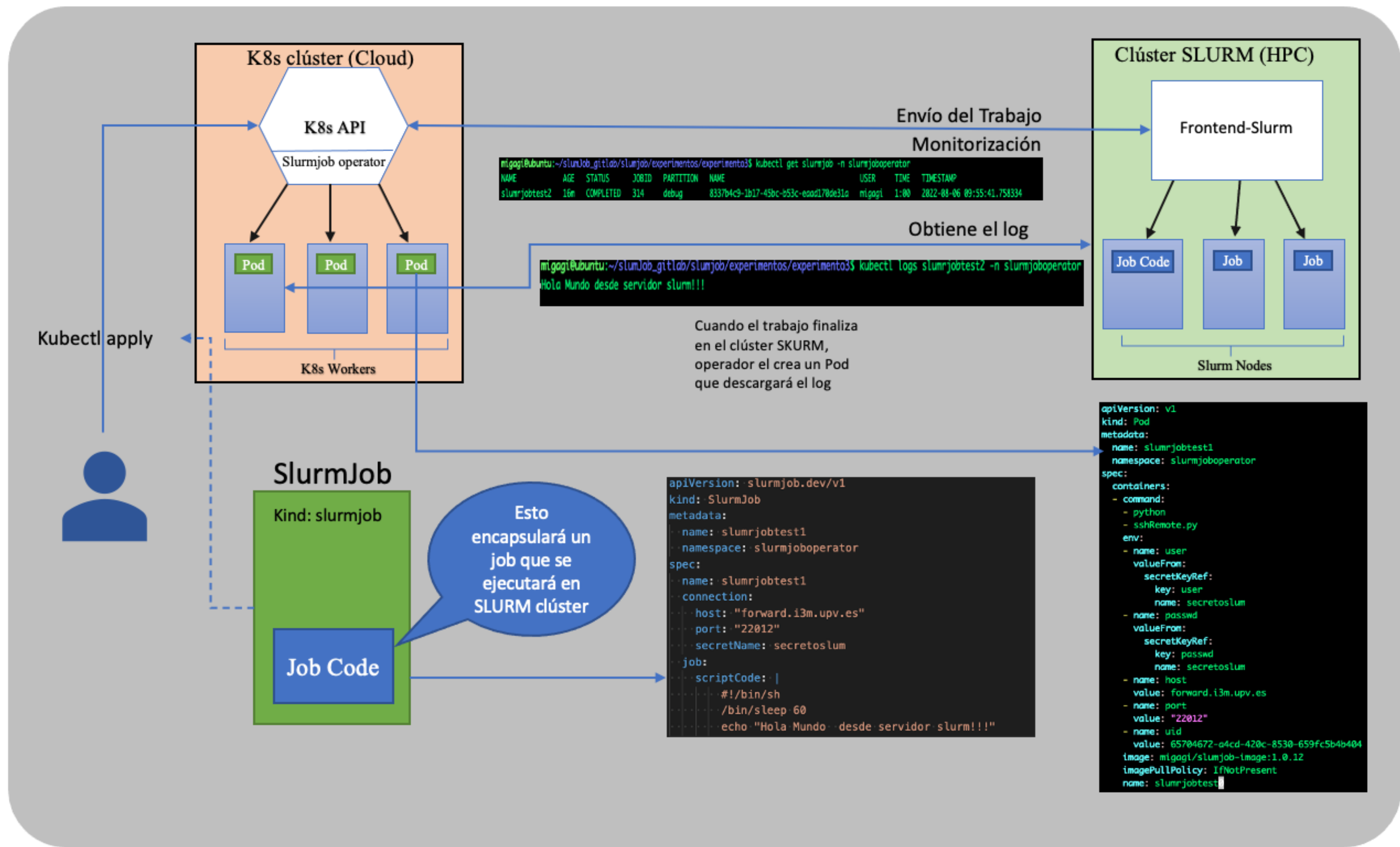

Imagen 7. Diagrama de arquitectura del sistema.

Componentes de Kubernetes

En esta sección se describe la implementación del diseño propuesto.

Todo el código del proyecto está desarrollado en código abierto y está subido en un repositorio de código en Gitlab. La URL para acceder al contenido del trabajo es: https://gitlab.com/migagi/slumjob.

Para poder implementar resolviendo todos los objetivos planteados, se tiene que crear un custom resource definition (*CRD*) con la definición de la información a manejar por el *operador* con el nuevo objeto que controlaremos con el *operador*. Hay que proporcionar un nombre al *CRD*, que en nuestro caso será *slurmJob.* Es importante definir el nombre y grupo en el *CRD*, ya que se usará en la API REST para indicar la ruta: /apis/<group>/<version>, donde K8s definirá todas las operaciones para manejar el objeto.

Más tarde, en la sección de seguridad utilizaremos el grupo definido para crear un *ClusterRole* que permita el acceso al nuevo recurso.

En el fichero *YAML*, en la figura 2 contiene un ejemplo de una definición de un Slurmjob que puede usarse como ejemplo, donde se observa que se indica la información de entrada que necesita el *operador* para conectarse al clúster SLURM y ejecutar el trabajo:

```
name: Nombre del Objeto
connection: Contendrá la información para la conexión con el servidor slurm
  host: Servidor
  port: puerto
  secretName: Nombre del objeto Secret con las credenciales
job:
  scriptCode: código del script a ejecutar
```

También hay que crear los campos con la información que se obtiene del clúster SLURM (jobid, partition, name, user, state y time). Esta información es la que proporcionaremos al usuario cuando utilice el comando `kubectl get slurmjob XX`. Para hacer esto posible se añaden las columnas en la sección *additionalPrinterColumns*.

Se han definido todos los parámetros como obligatorios, ya que todos se necesitan para poder ejecutar el trabajo correctamente. Para indicar que los parámetros son obligatorios, en la propia definición del objeto, en el atributo *required* se indica con el nombre de las columnas:

```
required:
    - host
    - port
    - secretName
```

A continuación, se muestra el fichero *YAML* con la definición del *CRD*:


```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
 name: slurmjobs.slurmjob.dev
spec:
 scope: Namespaced
 group: slurmjob.dev
 names:
  kind: SlurmJob
  plural: slurmjobs
  singular: slurmjob
  shortNames:
   - sljs
   - slj
 versions:
  - name: v1
   served: true
   storage: true
   schema:
    openAPIV3Schema:
     type: object
     properties:
      spec:
       type: object
       x-kubernetes-preserve-unknown-fields: true
       properties:
        name:
         type: string
         minimum: 1
        connection:
         type: object
         x-kubernetes-preserve-unknown-fields: true
         properties:
          host:
           type: string
          port:
           type: string
          secretName:
           type: string
         required:
          - host
          - port
          - secretName
        job:
         type: object
         x-kubernetes-preserve-unknown-fields: true
         properties:
          scriptCode:
           type: string
       required:
        - name
        - connection
        - job
      status:
       type: object
       x-kubernetes-preserve-unknown-fields: true
```

```yaml
additionalPrinterColumns:
 - name: Age
   type: date
   jsonPath:. metadata.creationTimestamp
 - name: status
   type: string
   description: The status slurm job sented to the cluster.
   jsonPath:. spec.status
 - name: jobid
   type: string
   description: The jobid slurm job.
   jsonPath: .spec.jobid
 - name: partition
   type: string
   description: The partition slurm job.
   jsonPath: .spec.partition
 - name: name
   type: string
   description: The name slurm job.
   jsonPath: .spec.name
 - name: user
   type: string
   description: The user slurm job.
   jsonPath: .spec.user
 - name: time
   type: string
   description: The time slurm job.
   jsonPath: .spec.time
 - name: timestamp
   type: string
   description: The timestamp slurm job.
   jsonPath: .spec.timestamp
```

Figura 1: Enlace al fichero con la definición del CRD.

Una vez creado el *CRD* necesitamos aplicar las políticas de seguridad *RBAC* para que el *operador* pueda acceder y actuar a los objetos de K8s necesarios.

En primer lugar, se crea un objeto *ServiceAccount* que se asigna al *operador* en el momento del despliegue. El objeto *ServiceAccount* sirve para identificar a un usuario concreto.

Se define el *ClusterRole* con todos los verbos (operaciones que realiza la API según la especificación de K8s: list, watch, patch, get) y recursos sobre los que el *operador* necesita tener permisos, sobre el grupo que se ha definido en el *CRD* (slurmjob.dev) se crea el *Role* indicando los objetos que se manejan *Pods* y *Secrets*.

Con el *ClusterRoleBinding* se asigna el *ClusterRole* al *ServiceAccount* definido y con el *RoleBinding* se asigna el *Role* creado al *ServiceAccount*, con esto se le proporciona al *operador* los permisos para trabajar con objetos de K8s. [enlace al fichero de configuración].

Hasta esta parte se describe todo lo necesario para que el *operador* pueda funcionar correctamente en K8s. A continuación, se explicará la implementación del *operador*.

Para el correcto funcionamiento del objeto slurmJob, el usuario debe definir un objeto *Secret* (objeto de K8s que permite el almacenamiento de datos sensibles) con el usuario y contraseña del clúster SLURM.

Los datos en un objeto *Secret* deben estar codificados en base64:

```
apiVersion: v1
kind: Secret
metadata:
 name: secretoslum
 namespace: slurmjoboperator
type: Opaque
data:
 user: bWlnYWdp
 passwd: TVVDTkFQdGZtMjE=
```

Se ha elegido implementar el *operador* con el framework de código abierto Kofp[10]. Implementado en Python, este framework proporciona las herramientas necesarias para comunicarse con K8s y trasformar la respuesta en objectos Python, proporcionando librerías internas muy básicas para manipular los objetos de K8s. En este desarrollo se ha utilizado una librería cliente K8s de python (*pykube-ng)* para crear el *Pod*, eliminarlo y recuperar el objeto *Secret*, ya que el framework no lo permite directamente.

Para que el *operador* pueda controlar los eventos que se producen en los objetos slurmJob, con el framework Kopf se dispone de handlers con los que manejar los eventos provocados por las operaciones crear, actualizar y borrar que se pueden realizar en K8s con el comando `kubectl`.

Se han definido tres *handlers*: *on_create*, *on_update* y *on_delete* los cuales manejan los eventos respectivos:

Operación de creación (on_create)

Cuando se produce un evento de creación del objeto slurmJob, el *handler on_create* se ejecuta. En primer lugar, el propio K8s valida que estén todos los datos obligatorios, pero no puede validar que el objeto Secret esté creado y que esté con el usuario y la contraseña definida. Esta validación se hace en el propio *operador* el cual utilizando la librería pykube recupera el objecto Secret y verifica que este definido el usuario y la contraseña.

Una vez tenemos los datos de entrada validados, el *operador* crea un fichero con el script a ejecutar en el clúster SLURM, el contenido de este es el que se encuentra en la columna scriptCode del objeto K8s, el fichero se nombrará con el UID (identificador único de K8s) del objecto K8s. Este se copia con el comando SCP al clúster SLURM y a continuación, por medio de SSH, se envía el comando de ejecución de trabajo batch (SBATCH) a la cola de ejecución.

Cuando se crea el fichero, se le añaden unas directivas que le indican a SLURM el nombre del trabajo, y el nombre del fichero de salida con el resultado del trabajo, en ambos casos se utiliza el UID del objeto K8s. Por ejemplo:

```
#!/bin/bash
#SBATCH -J "+uid+"\n")
#SBATCH --output='+uid+'.out\n')
```

Con la directiva `SBATCH -J` indicamos a SLURM el jobid del trabajo y con la directiva `SBATCH --output=` indicamos a SLURM el nombre del fichero output, en ambos casos utilizamos el UID del objeto K8s como jobid.

El tener el UID del objeto K8s como jobId nos facilita la forma de recuperar el estado de la ejecución en el clúster SLURM, de la misma forma cuando el trabajo finaliza, podemos recuperar el fichero de vuelta con el resultado, utilizando el UID del objeto.

El siguiente diagrama de flujo explica el comportamiento que hace el *operador* para crear el objeto.

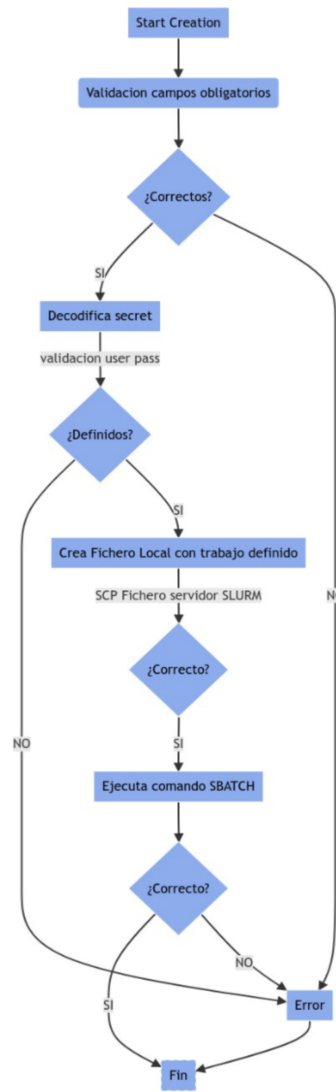

Imagen 8. Diagrama flujo operación creación.

Operación de actualización (on_update)

Solo se permite la modificación de un objeto que se encuentre en los estados (FAILED, COMPLETED o CANCELED), si el objeto está en estado COMPLETED, se elimina el *Pod* creado con el resultado de la anterior ejecución. En el resto de los estados no se realiza ninguna acción adicional. Si se actualiza un objeto en un estado que no sea alguno de los anteriores, simplemente se ignora el intento de cambio.

Cuando se produce un evento de actualización del objeto slurmJob, el *handler on_update* se ejecuta. En primer lugar, el propio K8s vuelve a validar que estén todos los datos obligatorios, pero al igual que en la creación, recupera el objeto Secret y verifica que este definido el usuario y la contraseña.

Al igual que en la creación se crea el fichero, se copia al clúster SLURM y se envía el trabajo con el comando de ejecución de trabajo batch (`SBATCH`) a la cola de ejecución.

El siguiente diagrama de flujo explica el comportamiento que hace el *operador* para actualizar el objeto.

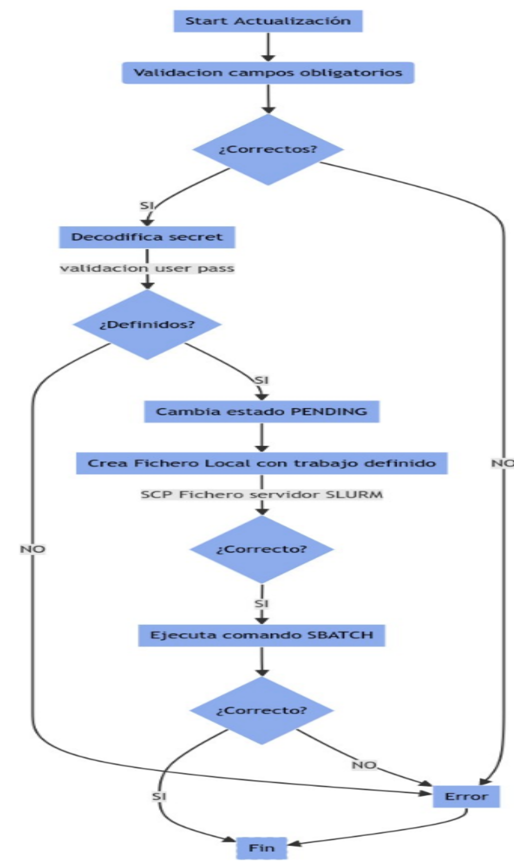

Imagen 9. Diagrama flujo operación actualización.

Operación de borrado (on_delete)

Cuando se produce un evento de eliminación del objeto slurmJob, el *handler on_delete* se ejecuta. Si el estado es COMPLETED, borrará el Pod creado con el resultado y el propio objeto. Si por el contrario el objeto está en estado RUNNING, en este caso mandará el comando `SCANCEL` por SSH para cancelar la ejecución en el clúster SLURM y posteriormente borrará el objeto.

En otros estados diferentes el objeto se borra sin más interacción.

El siguiente diagrama de flujo explica el comportamiento que hace el *operador* para borrar el objeto.

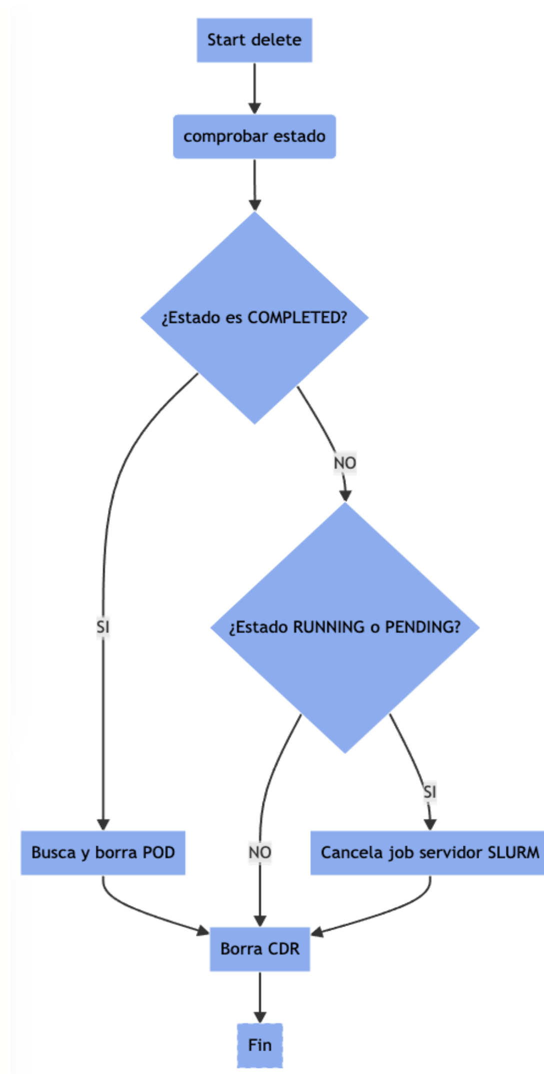

Imagen 10. Diagrama flujo operación delete.

La parte más importante del *operador* en la que se controla el ciclo de vida del objeto se ha desarrollado utilizando el manejador de ejecución *timer* que proporciona el framework Kopf, al cual se le define cada cuando tiempo se dispara. En nuestro caso se ha definido para que se dispare cada segundo.

Cuando un objeto *slurmJob* se crea, se inicia el *timer* y cuando se borra, el *timer* es borrado también.

Dentro del ciclo del *timer*, el *operador* se encarga de conectarse al clúster SLURM y ejecutar el comando `squeue` utilizando el UID del objeto K8s, para poder recuperar el estado del trabajo. En el cluster SLURM se ejecuta el siguiente comando:

```
squeue --format="%.18i;%.9P;%.40j;%.8u;%.9T;%.10M;" -n 622722d4-ad57-487c-99db-98b769be66a7 -t
all --sort=+i
```

Con el parámetro **-n** se define el JobId a recuperar. Por ejemplo, `squeue` -n *622722d4-ad57-487c-99db-98b769be66a7* obtendría el estado de la ejecución en SLURM del objeto K8s. Con el parámetro `format` **="%.18i;%.9P;%.40j;%.8u;%.9T;%.10M;"** indicamos que queremos recuperar las columnas (JOBID, PARTITION, NAME, USER, STATE y TIME), especificando el ancho de cada una las columnas y añadiendo un carácter punto y coma (";") para separar las columnas, lo que nos facilita procesar el resultado transformándolo en una colección de Python.

Un parámetro importante es `–sort=+i` que indica que nos devuelva los trabajos ordenados en función de su fecha de creación (del más reciente al más antiguo). De esta forma, si se ha ejecutado más de una vez el mismo trabajo (puede pasar con trabajos que tienen error) nos quedamos con la última instancia.

```
JOBID;PARTITION;                                     NAME;    USER;    STATE;      TIME;
  362;    debug;    622722d4-ad57-487c-99db-98b769be66a7;  migagi;COMPLETED;       1:00;
```

Imagen 11. Ejemplo de la salida del comando squeue.

Para poder realizar toda la comunicación es imprescindible obtener el objeto *Secret* y decodificar el usuario y contraseña que está codificada en base64. Para evitar tener que hacer esta acción cada segundo, se añade una cache en la que solo la primera vez se realiza esta acción y en las sucesivas se obtiene directamente de la cache, optimizando de esta forma el proceso.

En el siguiente diagrama se explica el funcionamiento:

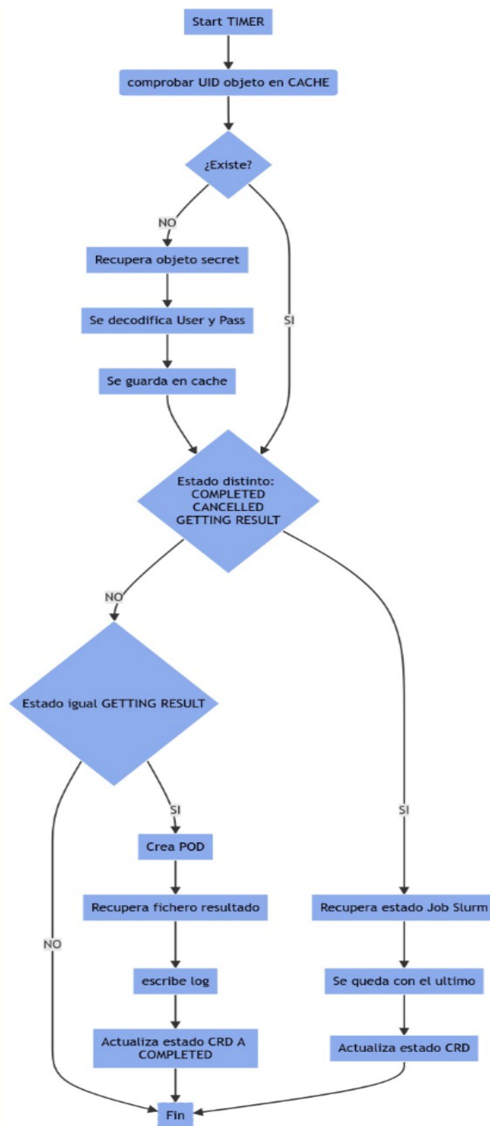

Imagen 12. Diagrama flujo timer.

Una vez tenemos decodificados los datos del usuario y contraseña, el *operador* se conecta por SSH al clúster SLURM recuperando el estado del trabajo y actualizando el estado del objeto K8s, quedando sincronizados. Si el estado recuperado es COMPLETED, se cambia el estado del objeto K8s a GETTING RESULTS, ya que a pesar de que el trabajo ha finalizado en el clúster SLURM, en K8s se tiene que crear un *Pod* con el mismo nombre del objeto *SlurmJob*. El *Pod* ejecuta un contenedor con una imagen con todo lo necesario para ejecutar código Python. Al contenedor se le pasa mediante variables de entorno todos los datos necesarios para conectarse al servidor y recuperar el fichero con el resultado del trabajo.

Entre las variables de entorno que recibe el *Pod* una de ellas es la del UID de objeto K8s. Como anteriormente indicábamos el nombre del fichero resultado tiene este UID como nombre.

De esta forma, el *Pod* se conecta al clúster SLURM, por SSH, donde ejecuta el comando `cat` al fichero con el output, utilizando la variable de entorno con el UID del objeto K8s localiza el fichero y lo imprime por la salida estándar. De esta forma se puede finalizar el *Pod*, quedando el resultado disponible en los logs.

La imagen esta subida en el repositorio de imágenes de Docker llamado Dockerhub con el nombre *migagi/slumjob-image:1.0.12*

El fichero YAML con la definición del *Pod* que se creará con cada objeto slurmjob para recuperar el fichero output:

```
apiVersion: v1
kind: Pod
metadata:
  name: slumrjobtest1
  namespace: slurmjoboperator
spec:
  containers:
  - command:
    - python
    - sshRemote.py
    env:
    - name: user
      valueFrom:
        secretKeyRef:
          key: user
          name: secretoslum
    - name: passwd
      valueFrom:
        secretKeyRef:
          key: passwd
          name: secretoslum
    - name: host
      value: forward.i3m.upv.es
    - name: port
      value: "22012"
    - name: uid
      value: 65704672-a4cd-420c-8530-659fc5b4b404
    image: migagi/slumjob-image:1.0.12
    imagePullPolicy: IfNotPresent
    name: slumrjobtest0
```

Utilizando esta definición el operador crear el Pod para recuperar el resultado, a continuación, se muestra el detalle de Pod creado por el operador:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl describe pod slumrjobtest1 -n slurmjoboperator
Name:         slumrjobtest1
Namespace:    slurmjoboperator
Priority:     0
Node:         minikube/192.168.49.2
Start Time:   Sun, 07 Aug 2022 11:30:07 +0000
Labels:       <none>
Annotations:  <none>
Status:       Succeeded
IP:           172.17.0.3
IPs:
  IP:           172.17.0.3
Controlled By:  SlurmJob/slumrjobtest1
Containers:
  slumrjobtest1:
    Container ID:  docker://810d25b4e6d0e0b27aec4c7f8cd84e43788b3fb1123bef44839fa56392bc130b
    Image:         migagi/slumjob-image:1.0.12
    Image ID:      docker-pullable://migagi/slumjob-image@sha256:59448fda47394f41d25e14e607f53386c1a1b62eab541ffe3a344832f4fc1099
    Port:          <none>
    Host Port:     <none>
    Command:
      python
      sshRemote.py
    State:          Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Sun, 07 Aug 2022 11:30:09 +0000
      Finished:     Sun, 07 Aug 2022 11:30:10 +0000
    Ready:          False
    Restart Count:  0
    Environment:
      user:    <set to the key 'user' in secret 'secretoslum'>    Optional: false
      passwd:  <set to the key 'passwd' in secret 'secretoslum'>  Optional: false
      host:    forward.i3m.upv.es
      port:    22012
      uid:     65704672-a4cd-420c-8530-659fc5b4b404
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-kvx5p (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
Volumes:
  kube-api-access-kvx5p:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  52s   default-scheduler  Successfully assigned slurmjoboperator/slumrjobtest1 to minikube
  Normal  Pulled     51s   kubelet            Container image "migagi/slumjob-image:1.0.12" already present on machine
  Normal  Created    50s   kubelet            Created container slumrjobtest1
  Normal  Started    50s   kubelet            Started container slumrjobtest1
```

Imagen 13. Detalle de un Pod con el resultado de un trabajo.

Despliegue

En este apartado vamos a describir los pasos necesarios para desplegar el *operador* en un clúster Kubernetes.

Primero hay que aplicar el fichero *YAML* con la definición del objeto SlurmJob en el *CRD*. El enlace al fichero en el repositorio es crd.yaml.

1. La instanciación del *CRD* mediante el comando `apply` sobre el fichero crd.yaml

```
kubectl apply -f crd.yaml
```

Antes de continuar con los siguientes pasos se tiene que crear el *namespace* donde estará ubicado el *Pod* con el *operador*.

2. Con el comando `create` se crea el *namespace slurmjoboperator*

```
kubectl create namespace slurmjoboperator
```

3. Crear *Service account*, *roles* y *rolebinding*.

   Para que el *operador* pueda funcionar correctamente se definen el *Service account*, los *roles* y el *rolebinding*. El enlace al fichero en el repositorio es rbac.yaml.

```
kubectl apply -f rbac.yaml
```

4. Crear el *Service* de tipo *ClusterIP*

   Creamos un *Service* tipo *ClusterIP* asignándolo al *deployment* especificando el puerto donde se recibirán las peticiones para validar los datos obligatorios del *Secret*.

   Necesitamos crear el servicio porque si no el *Pod* del *operador* permanece aislado y con el *Service*, K8s redirige las peticiones por el puerto indicado.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: slumjob-operator
  name: slumjob-operator-webhook
  namespace: slurmjoboperator
spec:
  type: ClusterIP
  ports:
    # Define metrics and profiling for them to be accessible within service meshes.
    - name: http-webhook
      port: 8443
      targetPort: 8443
      protocol: TCP
  selector:
    app: slumjob-operator
```

5. Crear el *deployment*

   Se define el fichero *YAML* con el *deployment* del *operador*, se crea un *Pod* en el *namespace* creado en el punto 2, el *Pod* tendrá el código del *operador* en un contenedor cuya imagen se llama *migagi/slumjob-operator:0.0.14* la imagen tiene que estar subida el registry, en nuestro caso utilizamos el que provee Docker hub. Enlace al fichero en el repositorio deployment.yaml

   En el anexo 1 se detalla cómo se ha creado la imagen del *operador* y se ha subido al registry.

La imagen creada y subida al registry de Docker hub es *migagi/slumjob-operator:0.0.14*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: slumjob-operator-deployment
  namespace: slumjoboperator
  labels:
    app: slumjob-operator
spec:
  replicas: 1
  selector:
    matchLabels:
      app: slumjob-operator
  template:
    metadata:
      labels:
        app: slumjob-operator
    spec:
      serviceAccountName: slumjoboperator-account
      containers:
      - name: slumjob-operator
        image: migagi/slumjob-operator:0.0.14
        ports:
        - containerPort: 80
```

6. Se despliega el objeto con el comando `apply`:

```
kubectl apply -f deployment.yaml
```

Después hay que verificar que el *operador* se ha desplegado correctamente.

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/deploy$ kubectl get pod -n slurmjoboperator
NAME                                          READY   STATUS    RESTARTS   AGE
slumjob-operator-deployment-79cf5bff6c-gtz8k  1/1     Running   0          20h
```

Imagen 14. Pod creado y en estado en ejecución.

7. Inspeccionamos el detalle del *Pod* con el comando `describe`.

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/deploy$ kubectl describe pod -n slurmjoboperator
Name:         slumjob-operator-deployment-79cf5bff6c-gtz8k
Namespace:    slurmjoboperator
Priority:     0
Node:         minikube/192.168.49.2
Start Time:   Sat, 06 Aug 2022 14:53:41 +0000
Labels:       app=slumjob-operator
              pod-template-hash=79cf5bff6c
Annotations:  <none>
Status:       Running
IP:           172.17.0.6
IPs:
  IP:           172.17.0.6
Controlled By:  ReplicaSet/slumjob-operator-deployment-79cf5bff6c
Containers:
  slumjob-operator:
    Container ID:   docker://3969303880b34446ada19898308a2e5c5845ce5c35c4d36733b3e33c568eaba3
    Image:          migagi/slumjob-operator:0.0.14
    Image ID:       docker-pullable://migagi/slumjob-operator@sha256:373837cfeae95b3c560048e27ff1c5cbe8cbba0595ac
    Ports:          80/TCP, 8443/TCP
    Host Ports:     0/TCP, 0/TCP
    State:          Running
      Started:      Sat, 06 Aug 2022 14:53:43 +0000
    Ready:          True
    Restart Count:  0
    Environment:
      SLURMJOB_OPERATOR_SERVICE_HOST:  10.97.6.218
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-4gsmd (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-4gsmd:
    Type:                    Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:                      <none>
```

Imagen 15. Salida comando `describe` con el detalle del Pod.

Una vez verificado que se ha desplegado correctamente, ya se pueden realizar los experimentos descritos en la siguiente sección.

Cuando el trabajo ha terminado, el *operador* crea un Pod para recuperar el trabajo del clúster SLURM, utilizando la imagen *migagi/slumjob-image:1.0.12* que tiene que estar subida al registry de Docker hub.

Validación y Pruebas

Esta sección describe en detalle, las validaciones realizadas con el fin de verificar el funcionamiento correcto de la implementación realizada del operador, validando que se cumplen todos los objetivos definidos.

Se han definido un conjunto de experimentos que además de validar los objetivos y la implementación, servirán para mostrar el funcionamiento del *operador*.

Entorno de pruebas

Para el entorno de pruebas se han utilizado dos servidores virtuales con los siguientes recursos:

Nombre Máquina	Tipo	Sistema Operativo	Memoria
Minikube_v1_25_2	Máquina Virtual	Ubuntu 18.04.5 LTS	11GB
SLURM_Server	Máquina Virtual	Ubuntu 18.04.5 LTS	6GB

La máquina virtual SlurmServer simula un clúster SLURM en el que el frontend también está configurado para ser también un nodo worker de SLURM. El clúster solo tiene recursos para ejecutar un trabajo simultáneamente, dejando el resto de los trabajos encolados en espera.

La máquina virtual Minikube representa un clúster Kubernetes con un solo nodo Máster y Worker, simplificando el entorno de pruebas.

Minikube[11] es una versión de Kubernetes que permite disponer de un entorno sencillo con la mayor parte de sus funciones, especialmente desarrollada para formación y validación.

Las dos máquinas virtuales se han desplegado en una infraestructura cloud gestionada mediante OpenNebula (plataforma cloud para administrar infraestructuras como servicio en recursos propios).

Experimentos

Para poder verificar el comportamiento esperado del *operador* implementado, se han diseñado una serie de experimentos que verificaran los objetivos planteados. En cada uno de los experimentos se verificará que el estado es el mismo tanto en el clúster SLURM como en K8s. Los experimentos verificarán los siguientes objetivos:

- Conseguir lanzar trabajos batch desde una máquina remota.
- Definir el ciclo de vida para la gestión de los trabajos desde K8s, (lanzar / cancelar / monitorizar).
- Extender K8s por medio de *CRD*.
- Capturar los trabajos fallidos notificando el resultado.
- Relanzar trabajos fallidos o relanzar trabajos completados.
- Obtener el resultado de un trabajo desde K8s.
- Implementar el sistema de manera no invasiva con respecto al clúster SLURM conectándose al mismo desde mediante un usuario regular.
- Realizar la implementación en código abierto.

Experimento 1: Trabajo simple con una salida por pantalla.

Con este experimento se valida el objetivo *"Conseguir lanzar trabajos batch desde una maquina remota"* al ejecutar un trabajo simple en el clúster SLURM, visualizando el resultado en los logs del Pod, de esta forma también cumplimos el objetivo *"Obtener el resultado de un trabajo desde K8s". A* continuación, se detalla el experimento cuyos resultados se han subido al repositorio para que puedan ser revisados.

Para la preparación del experimento vamos a definir un objeto Secret correcto, que tendrá las credenciales del usuario (usuario y contraseña) codificado en base64.

```yaml
apiVersion: v1
kind: Secret
metadata:
 name: secretoslum
 namespace: slurmjoboperator
type: Opaque
data:
 user: bWlnYWdp
 passwd: TVVDTkFQdGZtMjE=
```

Para crear el objeto en K8s se utiliza el comando:

```
kubectl apply -f nombre_fichero.yaml
```

Se ejecutará el siguiente script que simplemente ejecutará un comando `sleep` de 60 segundo he imprimirá por pantalla el mensaje "Hola Mundo desde servidor slurm!!!"

```
#!/bin/sh
/bin/sleep 60
echo "Hola Mundo desde servidor slurm!!!
```

Preparamos el fichero YAML con la definición del objeto SlurmJob:

```yaml
apiVersion: slurmjob.dev/v1
kind: SlurmJob
metadata:
  name: slumrjobtest1
  namespace: slurmjoboperator
spec:
  name: slumrjobtest1
  connection:
    host: "forward.i3m.upv.es"
    port: "22012"
    secretName: secretoslum
  job:
    scriptCode: |
      #!/bin/sh
      /bin/sleep 60
      echo "Hola Mundo desde servidor slurm!!!"
```

Figura 2: Fichero YAML con la definición de un objeto slurmjob.

Para crear el objeto en Kubernetes se utiliza el comando:

```
kubectl apply -f nombre_fichero.yaml
```

Después de ejecutar el comando, para comprobar que ya se ha creado el objeto, ejecutamos el comando:

```
kubectl get slurmjob -n slurmjoboperator
```

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl get slurmjob -n slurmjoboperator
NAME           AGE   STATUS    JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest1   5s    PENDING   312     debug       ecfa86b4-4ec1-489f-8b5e-cc3353e4faee   migagi   0:01   2022-08-06 09:04:50.537627
```

Imagen 16. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Cuando el trabajo ha terminado en el clúster SLURM el estado del objeto K8s cambia a “GETTING RESULT”:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl get slurmjob -n slurmjoboperator
NAME           AGE   STATUS           JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest1   65s   GETTING RESULT   312     debug       ecfa86b4-4ec1-489f-8b5e-cc3353e4faee   migagi   1:00   2022-08-06 09:05:50.601670
```

Imagen 17. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Como se ha explicado anteriormente, este estado representa que se está creando un Pod para recuperar el resultado del clúster SLURM y poder recuperarse posteriormente en los logs del Pod.

Cuando el objeto slurmJob ha finalizado:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl get slurmjob -n slurmjoboperator
NAME           AGE    STATUS      JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest1   117s   COMPLETED   312     debug       ecfa86b4-4ec1-489f-8b5e-cc3353e4faee   migagi   1:00   2022-08-06 09:05:55.683436
```

Imagen 18. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Vamos a ver el Pod que ha creado el operador:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl get pod slumrjobtest1 -n slurmjoboperator
NAME           READY   STATUS      RESTARTS   AGE
slumrjobtest1   0/1     Completed   0          23m
```

Imagen 19. Obtención de los Pod del namespace slurmjoboperator.

Como podemos ver, el trabajo ha finalizado y creando el Pod, que está en estado Completed, puesto que la única misión es obtener el fichero con el resultado e imprimirlo por la salida estándar, para que esté disponible en el log del Pod. Podemos ver el resultado consultando los logs del Pod conociendo su nombre del Pod (que es el mismo que el del objeto SlurmJob). Para ello podemos utilizar el siguiente comando:

```
kubectl logs slumrjobtest1 -n slurmjoboperator
```

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl logs slumrjobtest1 -n slurmjoboperator
Hola Mundo  desde servidor slurm!!!
```

Imagen 20. Obtención de los logs del Pod .

Podemos volcar el resultado del comando anterior en un fichero.

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl logs slumrjobtest1 -n slurmjoboperator > salidaExperimento1.txt
```

Imagen 21. Ejemplo comando para volcar logs en un fichero.

Después si eliminamos el objeto slurmJob, se debe borrar el Pod creado.

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl delete slurmjob slumrjobtest1  -n slurmjoboperator
slurmjob.slurmjob.dev "slumrjobtest1" deleted
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl get pod slumrjobtest1 -n slurmjoboperator
Error from server (NotFound): pods "slumrjobtest1" not found
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento1$ kubectl get slurmjob -n slurmjoboperator
No resources found in slurmjoboperator namespace.
```

Imagen 22. Borrado de un objeto slurmjob y obtención de los Pods del namespace slurmjoboperator.

Experimento 2: Trabajo con errores para su captura y notificación.

Con este experimento se valida el objetivo "Captura y notificación de un trabajo fallido". Además, se muestra la capacidad del operador para capturar el error mostrándolo cuando se consulta el estado del trabajo.

Para la ejecución de este experimento utilizaremos el mismo objeto Secret que en el experimento anterior. Para forzar el error en el trabajo vamos a hacer que el script a ejecutar tenga un comando inexistente (por ejemplo "slep" en vez de sleep):

```
#!/bin/sh
/bin/slep 600
echo "Hola Mundo desde servidor slurm!!!
```

Al igual que anteriormente se creará un fichero YAML con la definición del objeto SlurmJob.

Para crear el objeto en Kubernetes se utiliza el comando:

```
kubectl apply -f nombre_fichero.yaml
```

Después de ejecutar el comando, para comprobar que ya se ha creado el objeto ejecutaremos el comando:

```
kubectl get slurmjob -n slurmjoboperator
```

Después de pasado un instante, se puede comprobar que ha cambiado el estado del objeto, pasando a estar erróneo:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento2$ kubectl get slurmjob -n slurmjoboperator
NAME           AGE   STATUS   JOBID   PARTITION   NAME           USER   TIME   TIMESTAMP
slumrjobtest2  75s   FAILED                       slumrjobtest2                2022-08-06 09:46:08.779668
```

Imagen 23. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Podemos ver el trabajo ha finalizado con error.

Nos conectamos al clúster SLURM para comprobar que el estado es también fallido:

```
JOBID PARTITION                          NAME     USER      STATE      TIME
  313     debug 8337b4c9-1b17-45bc-b53c-eaad17   migagi    FAILED      0:00
```

Imagen 24. Salida comando estado job en cluster SLURM.

Como se muestra en la captura se observa que el estado es también fallido.

Experimento 3: Modificación de un trabajo con errores para su finalización correcta.

Con este experimento se valida la opción de que una vez subsanando un error y relanzado el trabajo, éste termina correctamente, mostrando la facilidad de ejecución del operador desarrollado.

Tomaremos el experimento anterior, modificado en el fichero YAML, el comando erróneo, y ejecutaremos el siguiente comando que actualizará el objeto en el lado del cloud, en K8s, relanzando el trabajo. De esta forma el estado pasará de FAILED a PEDING y a ejecutarse en el clúster SLURM.

Para crear el objeto en Kubernetes se utiliza el comando:

```
kubectl apply -f nombre_fichero.yaml
```

Después de ejecutar el comando, para comprobar que ya se ha creado el objeto ejecutamos el comando:

```
kubectl get slurmjob -n slurmjoboperator
```

Al corregirse el error ahora el trabajo SlurmJob está en ejecución en el clúster K8s:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento3$ kubectl get slurmjob -n slurmjoboperator
NAME           AGE    STATUS    JOBID  PARTITION  NAME                                  USER    TIME  TIMESTAMP
slumrjobtest2  8m42s  RUNNING   314    debug      8337b4c9-1b17-45bc-b53c-eaad170de31a  migagi  0:07  2022-08-06 09:54:43.896787
```

Imagen 25. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Cuando el trabajo termina:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento3$ kubectl get slurmjob -n slurmjoboperator
NAME           AGE  STATUS     JOBID  PARTITION  NAME                                  USER    TIME  TIMESTAMP
slumrjobtest2  16m  COMPLETED  314    debug      8337b4c9-1b17-45bc-b53c-eaad170de31a  migagi  1:00  2022-08-06 09:55:41.758334
```

Imagen 26. Obtención de los objetos slurmjob del namespace slurmjoboperator

Se comprueba que también termina en el clúster SLURM:

```
JOBID PARTITION                         NAME     USER      STATE       TIME
  314     debug 8337b4c9-1b17-45bc-b53c-eaad17   migagi  COMPLETED       1:00
```

Imagen 27. Salida comando estado job en cluster SLURM.

Para recuperar el resultado lo podemos recuperar en los logs del Pod:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento3$ kubectl logs slumrjobtest2 -n slurmjoboperator
Hola Mundo desde servidor slurm!!!
```

Imagen 28. Obtención de los logs del Pod.

La salida del trabajo está disponible en el repositorio.

Experimento 4: Modificación de un trabajo con estado COMPLETED para su relanzamiento

Con este experimento se valida la opción de volver a ejecutar un de un trabajo ya terminado correctamente, modificándolo antes mostrando la facilidad de ejecución del operador desarrollado.

Reutilizando el experimento anterior vamos a modificar la columna scriptCode:

```
job:
  scriptCode: |
      #!/bin/sh
      /bin/sleep 60
      echo "Experimento 4 modificacion trabajo COMPLETED!!!"
```

Imagen 29. Ejemplo columna scriptCode.

Al aplicarse el cambio, el objeto vuelve a ponerse en ejecución pasando del estado cambia de PENDING a RUNNING:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento4$ kubectl get slurmjob -n slurmjoboperator
NAME          AGE    STATUS    JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
 mrjobtest2   2m2s   RUNNING   347     debug       081812f3-3716-4bc5-8e60-f331c5881309   migagi   0:01   2022-08-06 13:50:50.853552
```

Imagen 30. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Pasado un tiempo verificamos en el clúster SLURM que el trabajo se ha completado. El trabajo que se ha completado es el 347 el 346 es el de la ejecución anterior:

```
JOBID PARTITION                         NAME           USER      STATE       TIME
  346     debug 081812f3-3716-4bc5-8e60-f331c5         migagi  COMPLETED       1:00
  347     debug 081812f3-3716-4bc5-8e60-f331c5         migagi  COMPLETED       1:00
```

Imagen 31. Salida comando estado job en cluster SLURM.

Con el mismo comando utilizado anteriormente recuperamos el fichero resultado de los logs del Pod:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento4$ kubectl logs slumrjobtest2 -n slurmjoboperator
Experimento 4 modificacion trabajo COMPLETED!!!
```

Imagen 32. Obtención de los logs del Pod.

Experimento 5: Ejecución de varios trabajos concurrentes de varios minutos de duración para crear contención y comprobación de las diferentes etapas por las que pasa.

En este experimento, queremos verificar que el operador es capaz de controlar varios trabajos concurrentes, de forma correcta, actualizado el estado en el objeto correspondiente.

Para ello creamos dos trabajos, el primero de una duración superior al segundo:

Trabajo 1, con nombre slurmjobtest5:

```
job:
  scriptCode: |
      #!/bin/sh
      /bin/sleep 120
      echo "Hola Mundo desde servidor slurm sleep 120!!!"
```

Imagen 33. Ejemplo columna scriptCode.

Trabajo 2, con nombre slurmjobtest5b:

```
job:
  scriptCode: |
      #!/bin/sh
      /bin/sleep 60
      echo "Hola Mundo desde servidor slurm sleep 60!!!"
```

Imagen 34. Ejemplo columna scriptCode.

Creamos el primer trabajo y el segundo:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento5$ kubectl get slurmjob -n slurmjoboperator
NAME            AGE   STATUS    JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest5   14s   RUNNING   348     debug       5d8b2ed2-d050-41e1-b04a-c00a8ccf7e37   migagi   0:09   2022-08-06 14:04:17.860651
slumrjobtest5b  9s    PENDING   349     debug       5ea7d7c5-cc8d-4756-9031-f49a937d8912   migagi   0:00   2022-08-06 14:04:16.953368
```

Imagen 35. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Como hemos ejecutado antes el comando con la creación del primer trabajo (slurmjobtest5) es el que está en estado RUNNING mientras que el segundo (slurmjobtest5b) tiene el estado PENDING, ya que el trabajo en el clúster SLURM esta esperado a que finalice la ejecución del primer trabajo.

Verificamos en el clúster SLURM que se corresponden los estados con los de K8s:

```
JOBID PARTITION                                NAME          USER     STATE       TIME
  348     debug 5d8b2ed2-d050-41e1-b04a-c00a8c              migagi   RUNNING     0:41
  349     debug 5ea7d7c5-cc8d-4756-9031-f49a93              migagi   PENDING     0:00
```

Imagen 36. Salida comando estado job en cluster SLURM.

Pasado dos minutos, que es lo que se había definido en el comando sleep, vemos que el trabajo uno ha finalizado (estado COMPLETED) y que, por tanto, el segundo está en ejecución:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento5$ kubectl get slurmjob -n slurmjoboperator
NAME            AGE     STATUS      JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest5   2m34s   COMPLETED   348     debug       5d8b2ed2-d050-41e1-b04a-c00a8ccf7e37   migagi   2:00   2022-08-06 14:06:14.109186
slumrjobtest5b  2m29s   RUNNING     349     debug       5ea7d7c5-cc8d-4756-9031-f49a937d8912   migagi   0:30   2022-08-06 14:06:38.338238
```

Imagen 37. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Verificamos en el clúster SLURM que se corresponden los estados con los de K8s:

```
JOBID PARTITION                          NAME                 USER      STATE       TIME
  348     debug 5d8b2ed2-d050-41e1-b04a-c00a8c              migagi  COMPLETED       2:00
  349     debug 5ea7d7c5-cc8d-4756-9031-f49a93              migagi    RUNNING       0:14
```

Imagen 38. Salida comando estado job en cluster SLURM.

Recuperamos el fichero resultado de los logs de ambos Pods:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento5$ kubectl logs slumrjobtest5 -n slurmjoboperator
Hola Mundo desde servidor slurm sleep 120!!!

migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento5$ kubectl logs slumrjobtest5b -n slurmjoboperator
Hola Mundo desde servidor slurm sleep 60!!!
```

Imagen 39. Obtención de los logs del Pod.

Experimento 6: Cancelar trabajo de larga duración en el clúster SLURM.

Con este experimento, se pretende validar que cuando hay en ejecución un trabajo de larga duración y se decide eliminarlo en el clúster SLURM se verifica que el operador actualiza el estado del objeto en K8s a CANCELED y en este caso se comprueba que pasa a ejecutarse el segundo trabajo que estaba pendiente.

Para ello crearemos dos trabajos:

Trabajo 1, con nombre slurmjobtest6:

```
job:
  scriptCode: |
    #!/bin/sh
    /bin/sleep 1200
    echo "Hola Mundo desde servidor slurm sleep 120!!!"
```

Imagen 40. Ejemplo columna scriptCode.

Trabajo 2, con nombre slurmjobtest6b:

```
job:
  scriptCode: |
    #!/bin/sh
    /bin/sleep 60
    echo "Hola Mundo desde servidor slurm sleep 60!!!"
```

Imagen 41. Ejemplo columna scriptCode.

Creamos el primer trabajo y el segundo:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento6$ kubectl get slurmjob -n slurmjoboperator
NAME            AGE   STATUS    JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest6   17s   RUNNING   356     debug       9ea59fdd-4586-4b32-9153-ac407cfc1f4b   migagi   0:11   2022-08-06 14:58:18.873470
slumrjobtest6b  9s    PENDING   357     debug       380f01ef-8a2a-4490-9431-85c810bcf5b9   migagi   0:00   2022-08-06 14:58:19.825417
```

Imagen 42. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Al igual que en el experimento anterior, como hemos ejecutado antes el comando con la creación del primer trabajo (slurmjobtest6) es el que está en estado RUNNING mientras que el segundo (slurmjobtest6b) tiene el estado PENDING.

Verificamos en el clúster SLURM, que se corresponden los estados con los de K8s:

```
JOBID PARTITION                          NAME              USER      STATE       TIME
  356     debug 9ea59fdd-4586-4b32-9153-ac407c             migagi    RUNNING     0:34
  357     debug 380f01ef-8a2a-4490-9431-85c810             migagi    PENDING     0:00
```

Imagen 43. Salida comando estado job en cluster SLURM.

Cancelamos el trabajo de larga duración, en el servidor SLURM, con el comando SCANCEL y el id del trabajo (en este caso es el 356), después de ejecutar el comando, se comprueba la lista de trabajos en ejecución y se verifica que el trabajo está en estado CANCELED:

```
JOBID PARTITION                          NAME              USER      STATE       TIME
  356     debug 9ea59fdd-4586-4b32-9153-ac407c             migagi  CANCELLED     0:42
  357     debug 380f01ef-8a2a-4490-9431-85c810             migagi    RUNNING     0:09
```

Imagen 44. Salida comando estado job en cluster SLURM.

En el clúster K8s tiene los mismos estados que en SLURM:

```
JOBID PARTITION                          NAME              USER      STATE       TIME
  356     debug 9ea59fdd-4586-4b32-9153-ac407c             migagi  CANCELLED     0:42
  357     debug 380f01ef-8a2a-4490-9431-85c810             migagi    RUNNING     0:09
```

Imagen 45. Salida comando estado job en cluster SLURM.

Al pasar un tiempo el trabajo finaliza correctamente:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento6$ kubectl get slurmjob -n slurmjoboperator
NAME             AGE    STATUS      JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest6    115s   CANCELLED   356     debug       9ea59fdd-4586-4b32-9153-ac407cfc1f4b   migagi   0:42   2022-08-06 14:58:50.789370
slumrjobtest6b   107s   COMPLETED   357     debug       380f01ef-8a2a-4490-9431-85c810bcf5b9   migagi   1:00   2022-08-06 14:59:57.526072
```

Imagen 46. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Recuperamos el fichero resultado del log del Pods:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento6$ kubectl logs slumrjobtest6b -n slurmjoboperator
Hola Mundo desde servidor slurm sleep 60!!!
```

Imagen 47. Obtención de los logs del Pod.

Experimento 7: Cancelar desde K8s el trabajo de larga duración, para que el trabajo de menor duración pase a ejecución.

Con este experimento, es igual al experimento 6, salvo que en este se elimina (equivalente a cancelar en el clúster SLURM) en K8s en vez de en el clúster SLURM.

Para ello crearemos dos trabajos:

Trabajo 1, con nombre slurmjobtest7:

```
job:
  scriptCode: |
      #!/bin/sh
      /bin/sleep 1200
      echo "Hola Mundo desde servidor slurm sleep 120!!!"
```

Imagen 48. Ejemplo colmna scriptCode.

Trabajo 2, con nombre slurmjobtest7b:

```
job:
  scriptCode: |
      #!/bin/sh
      /bin/sleep 60
      echo "Hola Mundo desde servidor slurm sleep 60!!!"
```

Imagen 49. Ejemplo columna scriptCode.

Creamos el primer trabajo y el segundo:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento7$ kubectl get slurmjob -n slurmjoboperator
NAME            AGE   STATUS    JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest7   19s   RUNNING   358     debug       20d939a4-bc3d-4ecf-8dc4-6d4532fb2fec   migagi   0:13   2022-08-06 15:22:29.000312
slumrjobtest7b  14s   PENDING   359     debug       50057ea8-6e09-48b2-84c5-7dbaf2a07595   migagi   0:00   2022-08-06 15:22:26.972697
```

Imagen 50. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Al igual que en el experimento anterior, como hemos ejecutado antes el comando con la creación del primer trabajo (slurmjobtest7) es el que está en estado RUNNING mientras que el segundo (slurmjobtest7b) tiene el estado PENDING.

Eliminamos desde K8s el trabajo número:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento7$ kubectl delete -f slurmjom.yaml
slurmjob.slurmjob.dev "slumrjobtest7" deleted
```

Imagen 51. Borrado objeto slurmjob.

Comprobamos que se ha cancelado el trabajo en el clúster SLURM tiene los mismos estados:

```
JOBID PARTITION                          NAME        USER     STATE      TIME
  358     debug 20d939a4-bc3d-4ecf-8dc4-6d4532     migagi  CANCELLED     0:30
  359     debug 50057ea8-6e09-48b2-84c5-7dbaf2     migagi    RUNNING     0:26
```

Imagen 52. Salida comando estado job en cluster SLURM.

Verificamos que el segundo trabajo este en ejecución:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento7$ kubectl get slurmjob -n slurmjoboperator
NAME            AGE   STATUS    JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest7b  34s   RUNNING   359     debug       50057ea8-6e09-48b2-84c5-7dbaf2a07595   migagi   0:02   2022-08-06 15:22:48.399471
```

Imagen 53. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Finalmente termina su ejecución:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento7$ kubectl get slurmjob -n slurmjoboperator
NAME            AGE    STATUS      JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest7b  109s   COMPLETED   359     debug       50057ea8-6e09-48b2-84c5-7dbaf2a07595   migagi   1:00   2022-08-06 15:23:53.290618
```

Imagen 54. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Experimento 8: Borrado de un trabajo en estado RUNNING en K8s.

Con este experimento, se pretende validar que cuando hay en ejecución un trabajo y se decide eliminarlo en K8s, se verifica que el operador cancela la ejecución en el clúster SLURM.

Creamos el trabajo y verificamos su estado:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento8$ kubectl get slurmjob -n slurmjoboperator
NAME            AGE   STATUS    JOBID   PARTITION   NAME                                   USER     TIME   TIMESTAMP
slumrjobtest8   10s   RUNNING   360     debug       3082d962-0435-4289-bb56-347c9f5f8aa4   migagi   0:04   2022-08-06 16:04:41.028676
```

Imagen 55. Obtención de los objetos slurmjob del namespace slurmjoboperator.

Eliminamos desde K8s el trabajo número uno:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento8$ kubectl delete -f slurmjom.yaml
slurmjob.slurmjob.dev "slumrjobtest8" deleted
```

Imagen 56. Borramos objeto slurmjob.

Comprobamos que se ha cancelado el trabajo en el clúster SLURM:

```
 JOBID PARTITION                           NAME                USER      STATE        TIME
   360     debug 3082d962-0435-4289-bb56-347c9f              migagi  CANCELLED        0:19
```

Imagen 57. Salida comando estado job en cluster SLURM.

Experimento 9: Validaciones de campos obligatorios y del objeto Secret.

Con este experimento, vamos a verificar que K8s valida los campos obligatorios (host, port, secretName y scriptCode), informando en caso contrario del campo que falta por rellenar, además se comprueba que el operador valida la existencia del objeto Secret y si este tiene el usuario y contraseña informado.

Se van a hacer un conjunto de pruebas verificando cada uno de los campos que tiene el objeto slurmJob.

- *Secret* con Usuario sin contraseña
  Se utiliza un objeto *Secret* sin contraseña, se tiene que verificar que devuelve el error que no tiene contraseña definida.
  Con el objeto *Secret* sin contraseña definido.

```
apiVersion: v1
kind: Secret
metadata:
 name: secretoslumtest
 namespace: slurmjoboperator
type: Opaque
data:
 user: bWlnYWdp
```

Se prepara un fichero *YAML* con la definición de un objeto *slurmJob* haciendo uso de un objeto *Secret* sin contraseña.

```yaml
apiVersion: slurmjob.dev/v1
kind: SlurmJob
metadata:
  name: slumrjobsecretuser
  namespace: slurmjoboperator
spec:
  name: slumrjobsecretuser
  connection:
    host: "test"
    port: "22012"
    secretName: secretoslumtest
  job:
    scriptCode: |
        #!/bin/sh
        /bin/sleep 120
        echo "Hola Mundo desde servidor slurm sleep 120!!!"
```

Creamos el trabajo y verificamos su estado:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento9$ kubectl apply -f slurmjom_secretSlum_user.yaml
Error from server: error when creating "slurmjom_secretSlum_user.yaml": admission webhook "check-numbers-ma.spec.connection.slurmjob.dev" denied the request: You must define S
ecret object with the User and password.
```

Imagen 58. Creación objeto slurmjob con salida indicando error.

Se verifica que no deja crear el trabajo y que informa que tiene que definir un objeto *Secret* con usuario y contraseña.

- *Secret* sin Usuario con contraseña

Se utiliza un objeto *Secret* sin user pero con contraseña, se tiene que verificar que devuelve el error que no tiene usuario definido.

```yaml
apiVersion: v1
kind: Secret
metadata:
 name: secretoslumtest2
 namespace: slurmjoboperator
type: Opaque
data:
passwd: TVVDTkFQdGZtMjE=
```

Se prepara un fichero *YAML* con la definición de un objeto *slurmJob* haciendo uso de un objeto *Secret*.

```yaml
apiVersion: slurmjob.dev/v1
kind: SlurmJob
metadata:
  name: slumrjobsecretpass
  namespace: slurmjoboperator
spec:
  name: slumrjobsecretpass
  connection:
    host: "test"
    port: "22012"
    secretName: secretoslumtest2
  job:
    scriptCode: |
        #!/bin/sh
        /bin/sleep 120
        echo "Hola Mundo desde servidor slurm sleep 120!!!"
```

Creamos el trabajo y verificamos su estado:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento9$ kubectl apply -f slurmjom_secretSlum_pass.yaml
Error from server: error when creating "slurmjom_secretSlum_pass.yaml": admission webhook "check-numbers-ma.spec.connection.slurmjob.dev" denied the request: You must define S
ecret object with the User and password.
```

Imagen 59. Creación objeto slurmjob con salida indicando error.

Se verifica que no deja crear el trabajo y que informa que tiene que definir un objeto *Secret* con usuario y contraseña.

- Sin el campo host
  Se prepara un fichero *YAML* con la definición de un objeto *slurmJob* sin definir el host.

```yaml
apiVersion: slurmjob.dev/v1
kind: SlurmJob
metadata:
  name: slumrjobhost
  namespace: slurmjoboperator
spec:
  name: slumrjobhost
  connection:
    host:
    port: "22012"
    secretName: secretoslum
  job:
    scriptCode: |
      #!/bin/sh
      /bin/sleep 60
      echo "Hola Mundo desde servidor slurm!!!"
```

Creamos el trabajo y verificamos su estado:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento9$ kubectl apply -f slurmjom_host.yaml
The SlurmJob "slumrjobhost" is invalid: spec.connection.host: Required value
```

Imagen 60. Creación objeto slurmjob con salida indicando error.

Se verifica que no deja crear el trabajo y que informa que falta por definir el campo host.

- Sin el campo port
  Se prepara un fichero *YAML* con la definición de un objeto *slurmJob* sin definir el port.

```yaml
apiVersion: slurmjob.dev/v1
kind: SlurmJob
metadata:
  name: slumrjobport
  namespace: slurmjoboperator
spec:
  name: slumrjobport
  connection:
    host: "test"
    port:
    secretName: secretoslum
  job:
    scriptCode: |
      #!/bin/sh
      /bin/sleep 60
      echo "Hola Mundo desde servidor slurm!!!"
```

Creamos el trabajo y verificamos su estado:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento9$ kubectl apply -f slurmjom_port.yaml
The SlurmJob "slumrjobport" is invalid: spec.connection.port: Required value
```

Imagen 61. Creación objeto slurmjob con salida indicando error.

Se verifica que no deja crear el trabajo y que informa que falta por definir el campo port.

- Sin el campo secretName

Se prepara un fichero *YAML* con la definición de un objeto *slurmJob* sin definir el secretName.

```yaml
apiVersion: slurmjob.dev/v1
kind: SlurmJob
metadata:
  name: slumrjobsecretname
  namespace: slurmjoboperator
spec:
  name: slumrjobsecretname
  connection:
    host: "test"
    port: "22012"
    secretName:
  job:
    scriptCode: |
      #!/bin/sh
      /bin/sleep 60
      echo "Hola Mundo desde servidor slurm!!!"
```

Creamos el trabajo y verificamos su estado:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento9$ kubectl apply -f slurmjom_secretName.yaml
The SlurmJob "slumrjobsecretname" is invalid: spec.connection.secretName: Required value
```

Imagen 62. Creación objeto slurmjob con salida indicando error.

Se verifica que no deja crear el trabajo y que informa que falta por definir el campo SecretName.

- Sin el campo scriptCode

Se prepara un fichero *YAML* con la definición de un objeto *slurmJob* sin definir el scriptCode.

```yaml
apiVersion: slurmjob.dev/v1
kind: SlurmJob
metadata:
  name: slumrjobscriptcode
  namespace: slurmjoboperator
spec:
  name: slumrjobscriptcode
  connection:
    host: "forward.i3m.upv.es"
    port: "22012"
    secretName: secretoslum
  job:
    scriptCode:
```

Creamos el trabajo y verificamos su estado:

```
migagi@ubuntu:~/slumJob_gitlab/slumjob/experimentos/experimento9$ kubectl apply -f slurmjom_scriptCode.yaml
Error from server: error when creating "slurmjom_scriptCode.yaml": admission webhook "check-numbers-ma.spec.connection.slurmjob.dev" denied the request: Script code must be set.
 Got None.
```

Imagen 63. Creación objeto slurmjob con salida indicando error.

Se verifica que no deja crear el trabajo y que informa que falta por definir el campo scriptCode.

Discusión de los resultados

Uno de los objetivos de la experimentación descrita en la sección anterior era verificar el cumplimiento de los objetivos planteados.

Como se puede ver en el experimento 1, se ha definido el caso de un trabajo batch que termina correctamente mostrando el *Pod* con el resultado y toda la información necesaria en el objeto K8s verificando el primer objetivo.

En el experimento 2 se ha forzado un error en la ejecución para comprobar que se captura el estado, verificando que se cumple el objetivo 3.

En los experimentos 3 y 4 se valida la característica del *operador* de permitir el relanzamiento, de trabajos fallidos o completados mostrando la flexibilidad del *operador*:

- En el experimento 3, se valida que modificando el fichero *YAML*, corrigiendo el problema provocado en el experimento 2 (un comando inexistente), y actualizando el objeto en K8S, se ejecuta y esta vez sí termina correctamente.

- En el experimento 4, hace referencia al relanzamiento de un trabajo que ya se ha completado, modificando el script, con las partes que se quiere modificar, actualizando el objeto en K8s se ejecuta en clúster SLURM.

Continuando con el experimento 5, donde se muestra como el *operador* es capaz de manejar el estado de dos trabajos concurrentes en K8s, pero ejecutados contra el mismo clúster SLURM el cual por su configuración, comprobamos que solo se permite la ejecución de un trabajo simultaneo, por lo que mientras no termine el trabajo en ejecución el otro se queda en estado PENDING y cuando termina el trabajo en ejecución, el trabajo en espera comienza a ejecutarse.

En los experimentos 6 y 7 se plantea la ejecución de dos trabajos simultáneos, como en el experimento 5, pero en estos experimentos se cancela el trabajo que está en estado RUNNING. En el experimento 6 se cancela desde el clúster SLURM y en el experimento 7 se cancela desde K8s, en ambos experimentos se verifica la correcta gestión de los estados. En el experimento 6 al cancelar el trabajo en SLURM se refleja en K8s y además se pone en ejecución el trabajo en espera, para el experimento 7 se borra el trabajo en K8s y se comprueba que se cancela el trabajo en el clúster SLURM al igual que en el caso anterior el trabajo pendiente se pone en ejecución.

En el experimento 7, se verifica que borrando un trabajo en estado RUNNING en K8s se cancela en el clúster SLURM.

En el último experimento se muestra que se verifica que el *operador* valida los campos obligatorios, informando en caso contrario de los campos por rellenar. Se verifica que el *operador* comprueba la existencia del objeto Secret y si este tiene el usuario y contraseña informado.

Como resultado de los experimentos se verifica que cumple con los objetivos marcados al inicio del TFM, verificando correcto funcionamiento del *operador*, mostrando además la funcionalidad.

Conclusiones

En el presente Trabajo Final de Máster se ha diseñado, implementado y validado un *operador* de K8s que automatiza la comunicación entre un clúster K8s y un clúster HTC con gestor de colas SLURM, con el presente trabajo se unen ambas tecnologías permitiendo gestionar trabajos por lotes (batch) de forma unificada desde K8s.

El presente *operador* proporciona simplicidad, bastando con disponer de acceso a un clúster HTC con un gestor de colas SLURM para utilizar este *operador*, permitiendo ejecutar trabajos en diferentes clústeres HTC desde un mismo clúster K8s.

Se han alcanzado todos los objetivos definidos respecto al modelo inicial, realizando una serie de experimentos (detallados en la sección anterior) que lo demuestran.

El trabajo se ha desarrollado combinando experiencias de diferentes áreas del máster, como la gestión de infraestructuras en la nube, la gestión de trabajos en colas y la gestión de contenedores, además de ser apropiado para tratar problemas que requieren computación de altas prestaciones.

El presente trabajo me ha servido como experiencia para ganar conocimiento en la gestión de contenedores, en el desarrollo de operadores y en la plataforma K8s.

Potenciales trabajos futuros:

- No utilizar *Secrets* para almacenar credenciales y utilizar componentes externos como Vault de Hashicorp en su lugar.
- Integración con sistemas de monitorización como son Prometheus y Grafana pudiendo definir cuadros de mando y la visualización de métricas como las ejecuciones de trabajos, seguimiento de estos y la definición de alertas.

Referencias

[1] K8s [Online] Available: http://kubernetes.io

[2] SLURM [Online] Available: https://slurm.schedmd.com/quickstart.html

[3] Kueue [Online] Available: https://github.com/kubernetes-sigs/kueue

[4] Docker container networking, [Online] Available: https://docs.docker.com/engine/userguide/networking/#default-networks

[5] Podman [Online] Available: https://podman.io/

[6] cri-o [Online] Available: https://cri-o.io/

[7] Singularity [Online] Available: https://docs.sylabs.io/guides/3.5/user-guide/introduction.html

[8] YAML [Online] Available: https://yaml.org/

[9] Staples, G. (2006). TORQUE resource manager. Conference on High Performance Networking and Computing: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing; 11-17 Nov. 2006, 8–es. https://doi.org/10.1145/1188455.1188464

[10] López-Huguet, S., Segrelles Quilis, J. D., Kasztelnik, M., Bubak, M., & Blanquer Espert, I. (2020). Seamlessly Managing HPC Workloads Through Kubernetes. Springer.

[11] Volcano [Online] Available: https://volcano.sh

[12] Kopf [Online] Available: https://kopf.readthedocs.io/en/stable/

[13] Minikube [Online] Available: https://minikube.sigs.k8s.io/docs/

[14] Containerd [Online] Available: https://containerd.io/

[15] Borg, Omega, and Kubernetes [Online] Available: https://queue.acm.org/detail.cfm?id=2898444

[16] N. Zhou, Y. Georgiou, L. Zhong, H. Zhou and M. Pospieszny, "Container Orchestration on HPC Systems," 2020 IEEE 13th International Conference on Cloud Computing (CLOUD), 2020, pp. 34-36, doi: 10.1109/CLOUD49709.2020.00017.

[17] Zhou, N., Georgiou, Y., Pospieszny, M. et al. Container orchestration on HPC systems through Kubernetes. J Cloud Comp 10, 16 (2021). https://doi.org/10.1186/s13677-021-00231-z

[18] Lublinsky, B., Jennings, E., & Spišaková, V. (2022). A Kubernetes "Bridge" operator between cloud and external resources.

[19] Objetivos y metas de desarrollo sostenible [Online] Available: https://www.un.org/sustainabledevelopment/es/sustainable-development-goals/

Apéndice 1. Creación imágenes *operador* y Pod

En este apéndice se muestra cómo se puede crear la imagen del contenedor Docker que contiene el *operador*.

Para la creación de la imagen hay que crear el fichero Dockerfile donde se especifican las instrucciones que se ejecutaran para construir la imagen.

Para el *operador* utilizamos la imagen Python:3.7 con lo necesario para ejecutar Python, además hay que instalar todas las librerías necesarias para la ejecución del *operador*.

Dockerfile

```
FROM python:3.7
LABEL maintainer=migagi@posgrado.upv.es
ARG kopf_version='1.35.4'

#RUN apt-get update && apt-get install -y python3 python3-pip
RUN pip install kopf==${kopf_version}
RUN pip install kopf[full-auth]
RUN pip install pykube-ng
RUN pip install requests
RUN pip install pyyaml
RUN pip3 install kubernetes
RUN pip install certbuilder
RUN pip install paramiko
#RUN apt-get update && apt-get install -y --no-install-recommends ntp
ADD  SSH.py /home/SSH.py
ADD slurmJob-operator.py /home/slurmJob-operator.py

CMD kopf run /home/slurmJob-operator.py --verbose --
liveness=http://0.0.0.0:8080/healthz
```

El código Python y la librería SSH.py se deja en el mismo directorio que el Dockerfile

Creamos la imagen:

```
docker build -t migagi/slumjob-operator:0.0.5 . --no-cache
```

Se sube al registry.

```
docker push migagi/slumjob-operator:0.0.5
```

Con la imagen subida ya se puede proceder al despliegue.

Apéndice 2. Relación con los Objetivos de Desarrollo Sostenible (ODS)

Los objetivos globales de desarrollo sostenible[19] se definieron por los líderes mundiales el 25 de septiembre de 2015, se adoptaron un conjunto de objetivos globales para erradicar la pobreza, proteger el planeta y asegurar la prosperidad para todos como parte de una nueva agenda.

Dentro de los 17 puntos definidos el presente trabajo fin de máster tiene relación con el punto 9 (Industria, Innovación e Infraestructura) ya se propone una herramienta que pretende integrar los existentes clústers HPC permitiendo seguir utilizando las herramientas existentes con las nuevas creadas en entornos cloud.

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. Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).