



UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INGENIERÍA
INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA
INFORMÁTICA

TRABAJO FIN DE MÁSTER

Diseño e implementación de un planificador en Kubernetes para
entornos de computación en la niebla de bajo coste

ALBERTO GÓMEZ GONZÁLEZ

Septiembre, 2023



UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INGENIERÍA
INFORMÁTICA

Departamento de Sistemas Informáticos

TRABAJO FIN DE MÁSTER

Diseño e implementación de un planificador en Kubernetes para
entornos de computación en la niebla de bajo coste

Autor: ALBERTO GÓMEZ GONZÁLEZ

Directoras: María del Carmen Carrión Espinosa
María Blanca Caminero Herráez

Septiembre, 2023

Resumen

La computación en la niebla surge como un complemento a la computación en la nube, que persigue acercar los recursos de cómputo y almacenamiento a los datos allí donde éstos se generan reduciendo tiempos de respuesta y optimizando el ancho de banda.

Por otro lado, han surgido los contenedores como modo de virtualización ligera creando una alternativa a las máquinas virtuales tradicionales. Kubernetes es actualmente la herramienta más extendida para la coordinación y gestión de estos contenedores.

Uno de los elementos clave de Kubernetes es su planificador, el cual se encarga de decidir donde se desplegarán los *pods*. Éstos son las unidades de computación desplegadas más pequeñas que se pueden crear y gestionar en Kubernetes y están formados por uno o varios contenedores.

Este Trabajo Fin de Máster se centra en estudiar el planificador de Kubernetes y extenderlo, ampliando su funcionalidad. Para ello se creará un entorno local que emula las características de un entorno de computación en la niebla, junto a otros componentes para obtener métricas *hardware* y del *kernel* del sistema operativo de los nodos del clúster y persistirla en una base de datos de series temporales. Con ello, el planificador desarrollado podrá realizar complejos cálculos para asignar los *pods* según la variación de estas métricas a lo largo del tiempo.

Otro apartado importante será el crear un nuevo recurso en Kubernetes que represente este planificador y un operador, que es un elemento que permite gestionar el ciclo de vida de estos recursos personalizados o *Custom Resources*.

A lo largo de este trabajo se verá todo el proceso que se ha llevado a cabo hasta alcanzar estos dos objetivos principales y además se comparará el funcionamiento de estos planificadores con otras opciones existentes en el mercado.

Abstract

Fog computing has emerged as a complement to cloud computing, which aims to bring computing and storage resources closer to the data where they are generated, reducing response times and optimizing bandwidth.

On the other hand, containers have emerged as a lightweight virtualization mode creating an alternative to traditional virtual machines. Kubernetes is currently the most widespread tool for the coordination and management of these containers.

One of the key elements of Kubernetes is its scheduler, which oversees deciding where pods will be deployed. Pods are the smallest deployable computing units that can be created and managed by Kubernetes and are formed by one or several containers.

This Master Thesis focuses on studying the Kubernetes scheduler and extending it, extending its functionality. To do so, a local environment that emulates the characteristics of a fog computing environment will be created, together with other components to obtain hardware and operating system kernel metrics from the cluster nodes and persist them in a time series database. With this, the developed scheduler will be able to perform complex calculations to allocate pods according to the variation of these metrics over time.

Another important section will be the creation of a new resource in Kubernetes that represents this scheduler and an operator, which is an element that allows managing the life cycle of Custom Resources.

Throughout this work we will see the whole process that has been carried out to achieve these two main objectives and we will also compare the operation of these schedulers with other existing options on the market.

Agradecimientos

En primer lugar, a mi mujer Lourdes por estar junto a mi durante todos estos años, apoyándome y dándome fuerzas para poder conseguir las metas que me he propuesto.

A mis padres, Antonio y Joaquina por todo el esfuerzo realizado durante tantos años para ofrecer todas las posibilidades que he tenido durante estos años y por seguir siendo un pilar tan importante en mi vida.

A mi hermano, Antonio por estar cerca todos estos años para lo bueno y lo malo y por ser un apoyo tan importante en mi vida.

A todos mis familiares y amigos, ya que, a pesar de nos está siempre cerca de ellos, nunca fallan cuando se les necesita.

Para acabar, me gustaría agradecer el apoyo y esfuerzo realizado por mis tutoras Carmen y Blanca, ya que sin ellas habría sido muy difícil poder finalizarlo.

Dedicado a Lourdes, Iker y Kira

.

ÍNDICE

CAPÍTULO 1. INTRODUCCIÓN.....	1
1.1 MOTIVACIÓN.....	1
1.2 OBJETIVOS	2
1.3 ESTRUCTURA DE LA MEMORIA	4
CAPÍTULO 2. ESTADO DEL ARTE	5
2.1 CONTENEDORES Y DOCKER	5
2.2 KUBERNETES	6
2.2.1 El patrón operador	9
2.2 PLANIFICACIÓN DE RECURSOS EN KUBERNETES	9
CAPÍTULO 3. METODOLOGÍA Y DESARROLLO	13
3.1 INTRODUCCIÓN	13
3.2 ENTORNO LOCAL DE KUBERNETES.....	15
3.2.1 Primera opción: Kind	16
3.2.2 Segunda opción: Minikube.....	17
3.3 SERVIDOR DE MÉTRICAS Y ANÁLISIS.....	17
3.3.1 Métricas de red	18
3.3.2 Métricas de disco	19
3.3.3 Métricas de CPU	20
3.4 PERSISTENCIA DE DATOS	20
3.4.1 Postgresql - <i>Custom adapter</i>	21
3.4.2 Timescaledb - Promscale.....	22
3.5 CREACIÓN DEL PLANIFICADOR	25
3.5.1 Descripción general	27
3.5.2 Parámetros de entrada.....	27
3.5.3 <i>Informers</i>	28
3.5.4 Filtrado por recursos.....	29
3.5.5 Filtrado por métricas mediante consultas a TimescaleDB	29
3.5.5.1 Consulta de un nivel.....	30
3.5.5.2 Consulta de dos niveles	32
3.5.6 Asignación de prioridades y elección de nodo	33
3.6 CREACIÓN DEL OPERADOR DE PLANIFICADORES	33
3.6.1 Kubebuilder y <i>scaffolding</i>	34
3.6.2 Definición del <i>Custom Resource</i>	35
3.6.3 <i>Controller</i> y objetos.....	36

CAPÍTULO 4. RESULTADOS	39
4.1 PRUEBA FUNCIONAL.....	39
4.2 EVALUACIÓN COMPARATIVA DE PRESTACIONES	42
4.2.1 Entorno de evaluación y carga aplicada.	43
4.2.2 Test de carga del clúster	44
4.2.3 Análisis de resultados	47
4.2.3.1 Escenario 1: Planificador por defecto	48
4.2.3.2 Escenario 2: Planificador aleatorio.....	49
4.2.3.3 Escenario 3: Planificador priorizando la asignación en nodos con bajo tráfico saliente de red.	50
4.2.3.4 Escenario 4: Planificador priorizando la asignación en nodos con mucho tiempo de <i>CPU</i> en espera.....	50
CAPÍTULO 5. CONCLUSIONES Y TRABAJO FUTURO.....	53
5.1 CONCLUSIONES	53
5.2 TRABAJO FUTURO Y POSIBLES AMPLIACIONES.....	54
BIBLIOGRAFÍA	57
LIBROS Y ARTICULOS	57
ENLACES INTERNET	58
GLOSARIO DE TÉRMINOS.....	63
ANEXO A. <i>SCRIPTS</i> DE INSTALACIÓN DE LOS COMPONENTES DEL CLÚSTER	65
ANEXO B. MATERIAL COMPLEMENTARIO	73

ÍNDICE DE FIGURAS

Figura 1. Diagrama de Gantt con la estimación inicial	3
Figura 2. Arquitectura y componentes de Kubernetes [3].....	7
Figura 3. Planificador dentro de la arquitectura de Kubernetes [20]	10
Figura 4. Visión general de la arquitectura desplegada.....	14
Figura 5. Arquitectura de la solución de persistencia de datos [28].....	22
Figura 6. Estructura de la vista <i>node_network_receive_bytes_total</i>	23
Figura 7. Resultados de <i>node_network_receive_bytes_total</i> con detalle.....	24
Figura 8. Valor máximo de la métrica <i>node_network_receive_bytes_total</i> por nodo	24
Figura 9. Diagrama de secuencia de creación de un <i>pod</i>	26
Figura 10. Resultado de consulta de un nivel.....	31
Figura 11. Entorno desplegado.....	39
Figura 12. <i>Pods</i> del <i>namespace monitoring</i>	40
Figura 13. Logs de <i>Promscale</i> ingestando métricas.....	40
Figura 14. Logs del operador de planificadores.	41
Figura 15. Log planificador.....	41
Figura 16. Logs del planificador tras lanzar el <i>pod</i>	42
Figura 17. <i>Pods</i> del <i>namespace ns1</i>	42
Figura 18. Despliegue del clúster de <i>Kafka</i>	45
Figura 19. Resultados de la consulta de transmisión de datos en cada nodo.	45
Figura 20. Ocupación tfm-dev-m03	46
Figura 21. Ocupación tfm-dev-m04	46
Figura 22. Ocupación tfm-dev-m05	46
Figura 23. Resultados de la consulta de CPUs en estado idle en cada nodo.....	47
Figura 24. E1: número de <i>Pods</i> en cada nodo	48
Figura 25. E1: Ejecución temporal de los <i>Pods</i>	48
Figura 26. E1: CPU consumida en cada nodo (mili).....	48
Figura 27. E1: Memoria consumida en cada nodo (KB).....	48
Figura 28. <i>Pods</i> no asignados por error <i>OutOfcpu</i>	49
Figura 29. E2: número de <i>Pods</i> en cada nodo	49
Figura 30. E2: Ejecución temporal de los <i>Pods</i>	49
Figura 31. E3: número de <i>Pods</i> en cada nodo	50
Figura 32. E3: ejecución temporal de los <i>Pods</i>	50
Figura 33. E3: número de <i>Pods</i> en cada nodo	51
Figura 34. E3: ejecución temporal de los <i>Pods</i>	51
Figura 35. Clúster de Kubernetes creado con Kind.....	66
Figura 36. Clúster de Kubernetes creado con Minikube	67
Figura 37. Servidor de monitorización desplegado.....	67
Figura 38. Resultado de consulta de métricas	68

ÍNDICE DE TABLAS

Tabla 1. Datos de la métrica <code>node_network_transmit_bytes_total</code>	16
Tabla 2: Modos de ejecución de la <i>CPU</i>	20
Tabla 3. Parámetros de entrada del planificador	28
Tabla 4. Parámetros para consulta de un nivel.	31
Tabla 5. Parámetros para consulta de dos niveles.	32
Tabla 6. Parámetros comunes de despliegue del planificador.....	35
Tabla 7. Recursos ocupados por nodo tras la instalación.....	43

ÍNDICE DE CÓDIGO

Código 1. Consulta de <i>node_network_receive_bytes_total</i> con detalle.....	24
Código 2. Consulta del valor máximo de la métrica <i>node_network_receive_bytes_total</i> por nodo.....	24
Código 3. Función de filtrado por recursos.....	30
Código 4. Consulta de un nivel	30
Código 5. Consulta de un nivel generada por el planificador	31
Código 6. Consulta de dos niveles.	32
Código 7. Consulta de dos niveles generada por el planificador	33
Código 8. <i>Script</i> de creación del API de MetricScheduler.....	35
Código 9. Descriptor CR MetricScheduler	36
Código 10. Descriptor del <i>pod</i> de prueba.....	41
Código 11. Consulta de transmisión de datos en cada nodo.	45
Código 12. Consulta de <i>CPUs</i> en estado <i>idle</i> en cada nodo.....	47
Código 13. Creación de registro local de Docker	65
Código 14. <i>Script</i> de creación del clúster mediante Kind	66
Código 15. <i>Script</i> de creación del entorno con Minikube	67
Código 16. Obtención de <i>Pods</i> del <i>namespace</i> de monitorización.....	67
Código 17. <i>Script</i> para obtener métricas	68
Código 18. Descriptor del <i>Deployment</i> del <i>adapter</i>	69
Código 19. Descriptor del <i>Service</i> del <i>adapter</i>	69
Código 20. <i>Script</i> de instalación de TimescaleDB y Promscale	70
Código 21. Descriptor del planificador para el escenario 3	71
Código 22. Descriptor del planificador para el escenario 4	72

ÍNDICE DE ECUACIONES

Ecuación 1. Ratio de errores de transmisión de datos	18
Ecuación 2. Tiempo medio de lectura en disco	19
Ecuación 3. Tiempo medio de escritura en disco	19
Ecuación 4. Valor medio de las CPUs de un servidor.....	20

CAPÍTULO 1. INTRODUCCIÓN

En esta primera sección de la memoria se introduce al lector a la motivación que ha llevado a realizar este trabajo, así como a los objetivos que se deberán cumplir y las capacidades que aplican a la realización de este. Por último, se enumerará brevemente los distintos capítulos de esta memoria.

1.1 MOTIVACIÓN

La computación en la niebla surge como un complemento a la computación en la nube, que persigue acercar los recursos de cómputo y almacenamiento a los datos allí donde éstos se generan. Esto permite reducir tiempos de respuesta y optimizar ancho de banda, entre otras mejoras, aspectos que resultan críticos en muchos casos de uso relacionados con Internet de las Cosas. Incluso si se pierde la conexión con la nube los dispositivos pueden seguir funcionando sin restricciones. Otra ventaja es que los datos no tienen por qué salir del lugar donde fueron creados, punto muy importante para adaptarse de manera más sencilla a las nuevas regulaciones de protección de datos.

Por otro lado, el uso de contenedores como mecanismo de virtualización ligera vive un auge y cada vez su uso es más extendido comiendo terreno a las máquinas virtuales tradicionales, debido en gran medida a la expansión de arquitecturas basadas en microservicios.

De ahí que sean necesarias herramientas que gestionen y coordinen el conjunto de contenedores que suponen la implementación de una aplicación o servicio, los orquestadores, de los cuales Kubernetes es el más popular. Uno de los elementos clave de Kubernetes es su planificador, encargado de decidir en qué recurso (nodo) se ejecutará cada trabajo (*pod*, que contiene uno o varios contenedores). Kubernetes ofrece mecanismos declarativos para influir en el comportamiento del planificador por defecto (tales como reglas de afinidad, tolerancias o requisitos de *CPU* y memoria), así como la posibilidad de incluir planificadores alternativos definidos por el usuario. Es posible particularizar en la especificación del contenedor los parámetros del planificador a usar e incluso utilizar varios planificadores en un mismo sistema.

1.2 OBJETIVOS

El objetivo general de este TFM es desarrollar y analizar las prestaciones de un nuevo planificador de aplicaciones para un clúster de bajas prestaciones, como puede ser un entorno local en un PC o un clúster de *Raspberry Pis*. En particular, se implementará un operador de Kubernetes capaz lanzar planificadores de aplicaciones que cumplan ciertas características que el usuario definirá y que estarán relacionadas con los valores de las métricas propias del clúster a nivel *hardware* y del *kernel* del sistema operativo. Se podrá estudiar la variación de estas métricas a lo largo de un periodo de tiempo.

Para el desarrollo de este TFM se hará uso de una metodología ágil basada en Scrum. Se mantendrán reuniones quincenales con las tutoras para el correcto seguimiento y retroalimentación del trabajo.

Otros subobjetivos que se deberán cumplir el desarrollo de este trabajo serán:

- Realizar un estudio funcional de Docker como principal herramienta para la creación de aplicaciones basadas en contenedores y de Kubernetes como estándar de facto de los orquestadores de estas aplicaciones.
- Estudiar cómo es la arquitectura de Kubernetes y sus interfaces de programación.
- Aprender cómo desarrollar nuevos recursos de Kubernetes y operadores que gestionen el ciclo de vida de estos recursos.

- Despliegue de un clúster con Kubernetes sobre un entorno local.
- Diseño y desarrollo de un nuevo recurso para planificar aplicaciones en el clúster.
- Implementación de un operador con el *framework* Kubebuilder que se encargue de lanzar estos planificadores.
- Evaluación de prestaciones del operador/planificador desarrollado y comparativa con otros existentes en el mercado.

En la Figura 1 mostramos mediante un diagrama de Gantt el tiempo invertido en la realización de las distintas tareas de este proyecto.

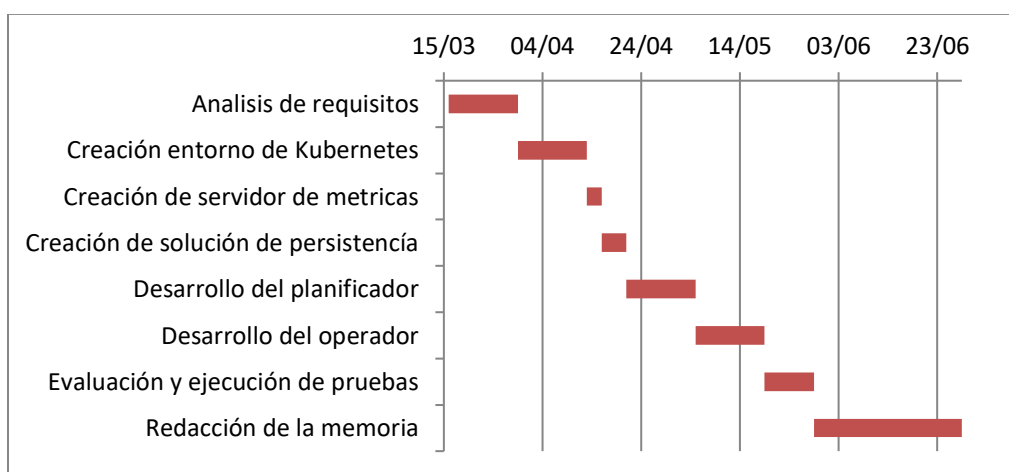


Figura 1. Diagrama de Gantt con la estimación inicial

Por otro lado, las competencias que se aplican son:

- CE1: Capacidad para la integración de tecnologías, aplicaciones, servicios y sistemas propios de la Ingeniería Informática, con carácter generalista, y en contextos más amplios y multidisciplinares.
- CE4: Capacidad para modelar, diseñar, definir la arquitectura, implantar, gestionar, operar, administrar y mantener aplicaciones, redes, sistemas, servicios y contenidos informáticos.
- CE09: Capacidad para diseñar y evaluar sistemas operativos y servidores, y aplicaciones y sistemas basados en computación distribuida.
- CE12: Capacidad para aplicar métodos matemáticos, estadísticos y de inteligencia artificial para modelar, diseñar y desarrollar aplicaciones, servicios, sistemas inteligentes y sistemas basados en el conocimiento.

- CE16: Realización, presentación y defensa, una vez obtenidos todos los créditos del plan de estudios, de un ejercicio original realizado individualmente ante un tribunal universitario, consistente en un proyecto integral de Ingeniería en Informática de naturaleza profesional en el que se sinteticen las competencias adquiridas en las enseñanzas.

1.3 ESTRUCTURA DE LA MEMORIA

En este apartado se muestra la disposición de los capítulos de la presente memoria con una breve descripción para que el lector obtenga una visión global del mismo.

Capítulo 1: Introducción. En este primer capítulo se introducirá al lector en el contexto de la herramienta, así como en la problemática en la que se centra. Además, se comentará la motivación que ha llevado a la realización de éste y los objetivos que se desean cubrir en el presente TFM.

Capítulo 2: Estado del arte. En este capítulo se introducirán conceptos relacionados con el tema del trabajo, con el objetivo de ofrecer una visión global sobre el mismo. Además, se describirán las herramientas utilizadas para su desarrollo.

Capítulo 3: Metodología y desarrollo. El tercer capítulo describe la metodología de trabajo llevada a cabo para realizar el trabajo, así como una explicación pormenorizada de todo el proceso de análisis, diseño e implementación de la solución realizada.

Capítulo 4: Resultados. En este capítulo se comprobará el funcionamiento e interacción de los distintos componentes diseñados en nuestro planificador y se realizará una evaluación comparativa frente a otros planificadores.

Capítulo 5: Conclusiones y trabajo futuro. En el quinto capítulo se concluye el desarrollo de este Trabajo Fin de Máster. En él se explicarán las conclusiones más importantes que se pueden extraer del desarrollo de este. Además, se detallarán posibles líneas de futuros desarrollos con los que mejorar el producto.

CAPÍTULO 2. ESTADO DEL ARTE

El objetivo de este TFM es desarrollar nuevas estrategias de planificación de contenedores en un clúster de Kubernetes. Para ello, en este capítulo definiremos conceptos clave como contenedores, Kubernetes, operador de desarrollo, métricas y bases de datos basadas en series temporales. También se describirán brevemente las herramientas utilizadas y se abordará la planificación de tareas en Kubernetes y el valor añadido del proyecto.

2.1 CONTENEDORES Y DOCKER

Para explicar lo que es Docker [Tur14][Nic19], previamente debemos detallar qué es un contenedor. Un contenedor es una forma de virtualización de un programa, ya sea desde un pequeño microservicio a un software con mayor entidad. Se diferencia de otros tipos de virtualización como las máquinas virtuales en que es mucho más ligero al no necesitar de un sistema operativo completo para su ejecución. Además, es portable y permite la recuperación ante fallos mucho más rápida que una máquina virtual tradicional.

Otro concepto que debemos conocer es el de imagen, que es un archivo que se encuentra compuesto de diversas capas y que se utiliza con el objetivo de ejecutar código de un contenedor de Docker. Una imagen actúa como un conjunto de instrucciones para generar un contenedor, como si fuera una plantilla.

Docker es una tecnología que facilita la creación y uso de contenedores. A continuación, se detallan algunas de sus ventajas:

- **Modularidad:** El enfoque de Docker para la creación de contenedores se basa en la posibilidad de poder eliminar pequeñas partes de la aplicación para mejorarlas o repararlas, sin tener que eliminarla completamente.
- **Control de versiones:** Cada imagen de Docker está formada por una serie de capas. Cada vez que se produce un cambio, se van formando nuevas capas, lo que aporta un control de los nuevos cambios añadidos. Estas capas se comparten entre distintos contenedores, lo que mejora su velocidad de creación y tamaño.
- **Rollback:** Una gran ventaja de esta arquitectura formada por capas es que podemos volver fácilmente a un estado anterior.
- **Despliegue rápido:** Facilita mucho el despliegue de aplicaciones, ya que una vez definido el sistema, puede ser instalado en cuestión de segundos.

Una aplicación de complejidad media-alta puede necesitar de varios contenedores o de varias instancias del mismo contenedor (por ejemplo, en aplicaciones configuradas con alta disponibilidad). La gestión de este tipo de aplicaciones puede llegar a ser muy compleja y es en este punto donde surge Kubernetes, facilitando muchas de estas tareas a los administradores de un clúster

2.2 KUBERNETES

Kubernetes [Say17] es una plataforma de orquestación de contenedores de código abierto, que automatiza muchos de los procesos manuales de despliegue, gestión y escalado de aplicaciones contenerizadas. Algunas de sus ventajas son:

- **Escalabilidad y balanceo de cargas:** se trata de una de las ventajas más importantes, ya que automatiza procesos complejos de despliegue y escalado de aplicaciones de forma automática.
- **Alta disponibilidad:** proporciona herramientas para garantizar la continuidad y la resistencia a fallos de las aplicaciones frente a posibles errores en el clúster.
- **Despliegue tanto en nubes públicas como privadas:** ofrece procesos para facilitar el despliegue tanto en local como en entornos *clouds*.

- **Comunidad amplia:** es una tecnología en continuo desarrollo con una comunidad enorme detrás que va añadiendo nuevas capacidades. Pertenece a la *Cloud Native Computing* [1].

Actualmente, se puede considerar como el estándar de facto en plataformas de orquestación de contenedores, ya que otras como Apache Mesos fueron discontinuadas [Car22].

Un concepto importante que debemos conocer, ya que utilizaremos infinidad de ocasiones de ahora en adelante, es el de *pod* [2], el cual es la unidad de computación más pequeña que se puede crear y gestionar en Kubernetes. Es un grupo de uno o más contenedores, con almacenamiento/red compartidos, y unas especificaciones de cómo ejecutar los contenedores. Se considera la unidad más pequeña de las aplicaciones de Kubernetes.

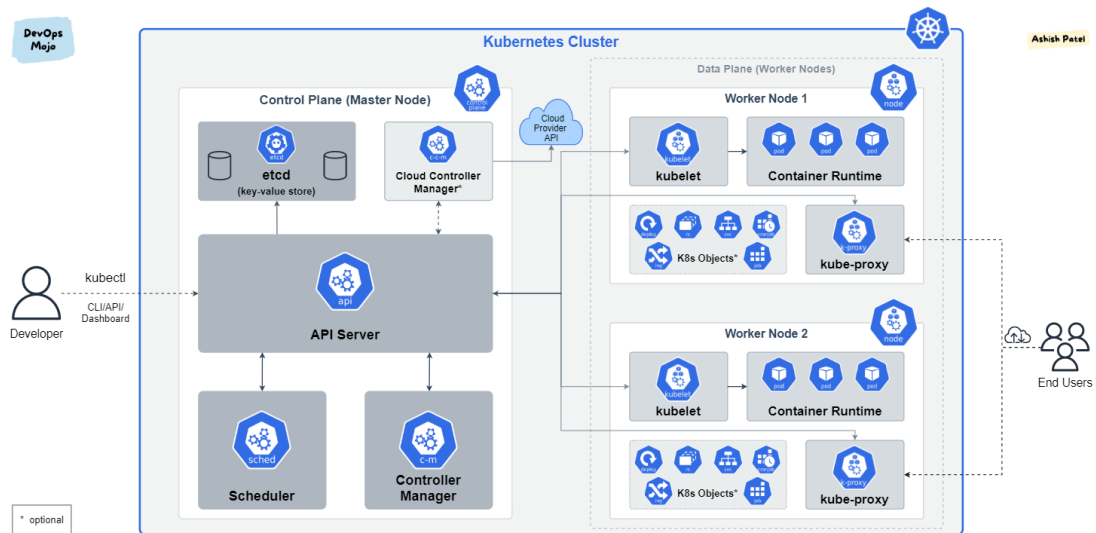


Figura 2. Arquitectura y componentes de Kubernetes [3]

En la Figura 2. Arquitectura y componentes de Kubernetes podemos ver la arquitectura de Kubernetes y su modelo maestro-esclavo. El nodo maestro es el encargado de la administración del clúster y es el punto de entrada para tareas administrativas. Es posible configurar más de una instancia maestra ofreciendo de esta forma alta disponibilidad y siendo sólo uno de estos nodos líder. Algunos de los componentes que forman el nodo maestro o *control plane* son:

- *API server*: es un componente del *control plane* que expone la API de Kubernetes. Se puede considerar como el *front-end* del *control plane*.
- *Etcd*: se trata del almacén de datos donde Kubernetes guarda y replica todos los estados del clúster.
- *Scheduler* o planificador: es un proceso que se encarga de asignar los *pods* a los nodos. Determina que nodos son válidos para cada *pod* de acuerdo con una serie de restricciones y a la disponibilidad de recursos.

Por otro lado, tenemos los nodos esclavos o *workers*, que son los servidores donde se ejecutan las aplicaciones. Algunos de los componentes que nos encontramos en estos nodos son:

- *Kubelet*: es un agente que se ejecuta en los nodos de trabajo y se asegura que los *pods* estén en funcionamiento y en un estado saludable. No gestiona contenedores que no hayan sido creados mediante Kubernetes.
- *Container-runtime*: Es el entorno de ejecución de contenedores, como Docker o *containerd*, que se encarga de ejecutar y gestionar los contenedores en el nodo *worker*.

En Kubernetes existe el concepto de *controllers* [4], los cuales rastrean determinados tipos de recursos. Esos objetos tienen en todo momento un estado deseado. El *controller* es el responsable de que el estado actual de ese objeto sea lo más parecido posible al deseado. Un ejemplo de *controller* es un *ReplicaSet* [5] que garantiza un número específico de réplicas en un determinado momento. Otro ejemplo sería un *Deployment* [6] que es un concepto a más alto nivel al que gestiona *ReplicaSet* y que proporciona actualizaciones de forma declarativa de un *pod*. El *controller-manager* es un demonio que se encarga de gestionar estos *controllers*.

Kubernetes permite extender su funcionalidad creando nuevos *controllers*. Los operadores que usaremos en este trabajo permiten crear nuevos *controllers* como describiremos más adelante.

2.2.1 El patrón operador

Los operadores [Dob20] son una extensión de Kubernetes que hacen usos de *Custom Resources* que son extensiones del API de Kubernetes y que representan nuevas aplicaciones o componentes que el administrador podrá gestionar. Permiten ampliar las funcionalidades que ofrece Kubernetes y adaptarlo a las necesidades del desarrollado o administrador del clúster [Zei22]. Existen herramientas que facilitan el desarrollo de estos operadores como Operator SDK [7] y Kubebuilder [8]. De esta última, hablaremos más en profundidad en la sección 3.6.1.

2.2 PLANIFICACIÓN DE RECURSOS EN KUBERNETES

Como último apartado de este capítulo, se explicará brevemente cómo realiza nativamente Kubernetes la planificación de las aplicaciones y veremos las opciones que provee para crear planificadores personalizados.

El proceso de *scheduling* o planificación en Kubernetes [9][10] se refiere al proceso a través del cual se asegura que un determinado *pod* pueda ser asignado en un determinado nodo del clúster de modo que el *kubelet* correspondiente pueda ejecutarlo. En la Figura 3, podemos ver una visión general de dónde se sitúa el planificador dentro de la arquitectura de Kubernetes. En ella, podemos ver como el planificador monitorea el almacén de objetos en búsqueda de nuevos *pods* sin asignar, a los cuales asignará un posible nodo donde ser ejecutados. Entonces, a su vez el *kubelet* monitorizará el almacén de objetos en busca de *pods* asignados sin ejecutar, para ejecutarlos.

Profundizando en este proceso, ¿qué ocurre cuando un *pod* es creado en un clúster? De cara al usuario este proceso se completa en unos pocos segundos, pero en *background* está ocurriendo un proceso bastante complejo:

- Mientras que el *API server* está escaneando el clúster (lo hace continuamente), el planificador detecta que hay un nuevo *pod* que no tiene asignado el parámetro *nodeName*, el cual indica qué nodo es el propietario de este *pod*.

- El *scheduler* elige un nodo apropiado donde situar este *pod* y le asigna el nombre de este.
- El *kubelet* del nodo seleccionado es notificado de que hay un *pod* pendiente de ejecución.
- El *kubelet* del nodo seleccionado ejecuta el *pod*.

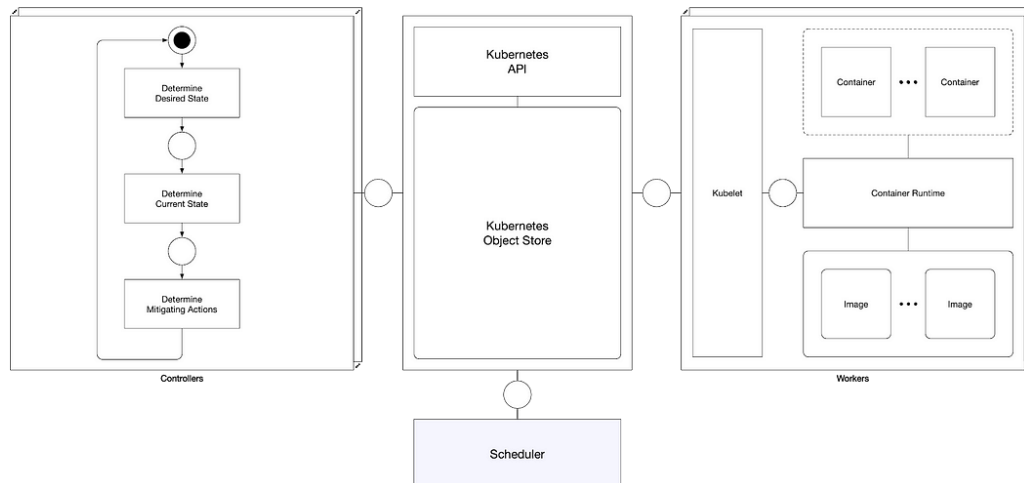


Figura 3. Planificador dentro de la arquitectura de Kubernetes [20]

¿Cómo elige el planificador el nodo correcto? Es la parte más compleja de todo este proceso. Para ello el planificador hace uso de varios algoritmos para tomar la decisión correcta. Algunos de estos algoritmos dependen de opciones seleccionadas por el usuario, mientras que otras las calcula el propio planificador.

El primer paso será comprobar si hay recursos de *CPU* y memoria suficiente en cada uno de los nodos para cumplir con los requisitos solicitados por el *pod* a ejecutar. En caso de no ser así, el nodo será automáticamente descartado. En caso de aplicaciones con estado también se comprobará que existe algún volumen de disco con las especificaciones solicitadas.

En este punto entra en juego el concepto de nodos factibles [11], que es un nuevo proceso de filtrado ante posibles restricciones definidas por el usuario. Algunas de estas restricciones son *NodeSelector*, en la que se añaden *labels* a la especificación del *pod* para restringir en qué nodos podrán ser asignados; *Affinities* o *Antiaffinities*, que son similares a *NodeSelector*, pero permiten asignar el *pod* a algún nodo incluso si no cumple con las

restricciones indicadas, lo que se considera una restricción *soft*. Por otro lado, los *Taints* seleccionan aquellos nodos que permiten "repeler" ciertos *pods*.

Una vez filtrados los nodos en los cuales hay recursos disponibles, se pasará a la etapa de *scoring* o puntuación. Esta puntuación se realiza basándose en el estado y la carga de los distintos nodos.

Una vez puntuados los nodos se asignará el nodo con mejor calificación y se comunicará al *kubelet* para que lo ejecute. En el caso de clústeres muy sobrecargados es posible que el *kubelet* no pueda ejecutar el *pod* debido a que haya cambiado el estado del nodo. En ese caso el planificador intentará seleccionar otro nodo.

Un clúster de Kubernetes admite uno o varios planificadores [12]. *Kube-scheduler*, el planificador de facto de Kubernetes, junto a las opciones de añadir restricciones por parte del usuario, es muy útil en la mayoría de los casos para realizar la planificación de nuestras aplicaciones. Sin embargo, en otros casos, el administrador puede querer disponer de un control más exhaustivo sobre el despliegue de sus *pods*. Por ejemplo, le podría interesar basarse en determinadas medidas obtenidas de componentes *hardware* o del *kernel* del sistema operativo para decidir donde asignar los *pods*.

Desde la propia comunidad se han realizado pruebas de concepto, creando nuevos planificadores que por ejemplo lanzan los *pods* de manera aleatoria [13]. Algunos de estos ejemplos han servido de base para entender cómo construir nuestro propio planificador.

CAPÍTULO 3. METODOLOGÍA Y DESARROLLO

En este capítulo, el más largo de esta memoria, se describirán todos los aspectos relacionados con el diseño llevado a cabo, los problemas, limitaciones y condicionantes encontrados durante su desarrollo, así como la metodología empleada.

3.1 INTRODUCCIÓN

La metodología utilizada para realizar el proyecto ha sido ágil con un enfoque sistemático y progresivo para realizar este trabajo. Se mantenían reuniones periódicas con las directoras en las que se iba mostrando el avance en el proyecto y se iba acordando con ellas por dónde continuar en las siguientes semanas, hasta alcanzar un punto óptimo y maduro en el desarrollo. El flujo de trabajo que se ha llevado a cabo para la realización del proyecto y del cual se va a profundizar a lo largo de este capítulo, ha seguido los siguientes puntos:

- **Entorno local de Kubernetes:** despliegue del entorno, los problemas encontrados y la justificación de la elección final de la herramienta para realizar el despliegue.
- **Despliegue del servidor de métricas:** qué herramientas se han utilizado para obtener las métricas del clúster.
- **Análisis de las métricas:** se discutirá sobre qué métricas son interesantes para usar por nuestro planificador.

- **Persistencia de datos:** se verán las opciones barajadas para realizar la persistencia de las métricas y se justificará la decisión adoptada.
- **Creación del planificador:** se profundizará en la creación del planificador que usaremos para desplegar las aplicaciones.
- **Creación del operador de planificadores:** Se detallará como se ha desarrollado el operador y los componentes que forman el mismo.

En la Figura 4 puede verse una visión general de la arquitectura de nuestro proyecto. En él podemos ver un servidor de métricas (Prometheus) que obtendrá periódicamente métricas de los nodos del clúster de Kubernetes. Entre ellas, métricas *hardware* y del *kernel* mediante el componente *node-exporter*. Por otro lado, existe un componente (Promscale) que cada cierto tiempo persistirá la información de las métricas en una base de datos (TimescaleDB).

Por último, vemos nuestro planificador (*o scheduler*), que obteniendo información del *API server* de Kubernetes a través de los *informers* [14] y realizando consultas a la base de datos, será capaz de elegir y asignar el nodo más adecuado a las necesidades del usuario para los nuevos *pods* a ejecutar en el sistema.

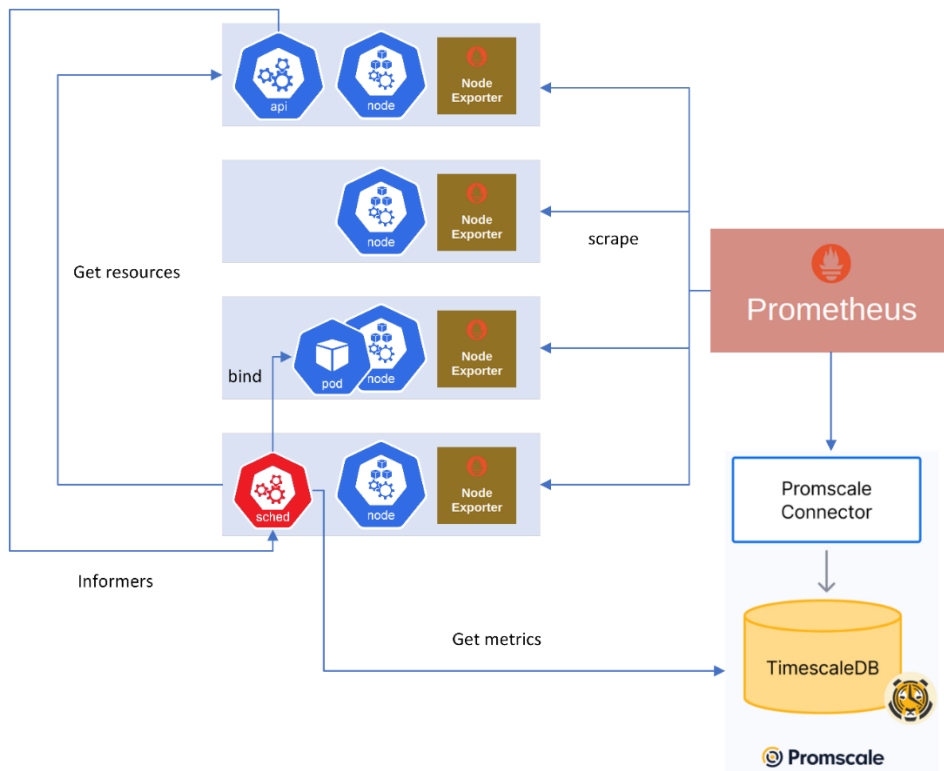


Figura 4. Visión general de la arquitectura desplegada.

Esta solución podría desplegarse de distintas formas, ya sea en un servidor en *cloud* o físico, en clúster montado con *Raspberry Pis*, en un ordenador mediante virtualización o despliegue de contenedores. En el siguiente apartado desarrollaremos las razones por las que se decidió utilizar un entorno local en este caso y profundizaremos en cada uno de los componentes que la forman.

3.2 ENTORNO LOCAL DE KUBERNETES

En el momento de afrontar este proyecto, una de las primeras decisiones que debía tomarse era donde se iba a crear el entorno de Kubernetes. En un primer momento, se pensó en realizar el despliegue en un *cloud*, pero esta opción se desechó para centrarnos en el objetivo del TFM y reducir los costes de desarrollo. También se barajó la posibilidad de utilizar un pequeño clúster de *Raspberry Pis*, de las cuales la Escuela dispone de un número importante de ellas.

Dado que el objetivo final del TFM es diseñar un nuevo planificador y se requiere de un entorno de desarrollo controlado que nos permita verificar el funcionamiento del planificador y poder probar rápidamente cambios que se apliquen al código. Al final, se optó por desplegar el clúster localmente ya que disponemos de un equipo bastante potente (*Dell Precision 5540* con *CPU i5-9400H* y *32 GB RAM*) que permite configurar rápidamente un entorno con cinco nodos (un *master* y cuatro *workers*). Cabe mencionar que todos los diseños realizados son fácilmente exportables a entornos de producción *cloud*, o *fog* basados en *Raspberry Pis*.

Con esta premisa, se buscaron herramientas que permitieran instalar un clúster de Kubernetes localmente sin necesidad de hacer uso de *clouds* o de tener que montar complejas arquitecturas con múltiples nodos.

3.2.1 Primera opción: Kind

Tras estudiar las distintas opciones para desplegar un clúster de Kubernetes se optó por la herramienta Kind [15]. Esta herramienta permite lanzar un clúster en local haciendo uso de contenedores Docker, creando un contenedor por cada uno de los nodos.

Es sencilla de configurar y muy liviana al levantar únicamente contenedores y no máquinas virtuales como otras opciones. Junto a la creación de los nodos, también se tuvo que crear un *registry* local de Docker en donde guardar las aplicaciones que posteriormente se crearían (planificador, operador, ...). En el Anexo A.2 se detallan los pasos para la instalación.

En las primeras fases del proyecto parecía que la decisión tomada había sido la correcta, pero en el punto en el cual ya estaba montado el servidor de métricas y se estaban realizando pruebas de carga sobre el sistema se observaban datos bastante confusos, ya que las métricas obtenidas en cada uno de los nodos eran prácticamente iguales, aumentando siempre el valor de cada una de las tomas. En la Tabla 1. Datos de la métrica *node_network_transmit_bytes_total*, se pueden ver los datos obtenidos de la métrica *node_network_transmit_bytes_total*, la cual mide el número de bytes transmitidos por la interfaz de red.

instance	metric_time	metric_value
10.244.1.3:9100	2023-05-03 18:02:18.403+00	5771586
10.244.2.4:9100	2023-05-03 18:02:20.15+00	5772949
10.244.4.2:9100	2023-05-03 18:02:21.669+00	5785410
10.244.5.2:9100	2023-05-03 18:02:21.684+00	5786762
10.244.1.3:9100	2023-05-03 18:02:23.403+00	5803872
10.244.2.4:9100	2023-05-03 18:02:25.15+00	5817787
10.244.4.2:9100	2023-05-03 18:02:26.669+00	5863623
10.244.5.2:9100	2023-05-03 18:02:26.684+00	5877733
10.244.1.3:9100	2023-05-03 18:02:18.403+00	5891818

Tabla 1. Datos de la métrica *node_network_transmit_bytes_total*

Estos datos son a todas luces incorrectos ya que la carga en cada uno de los nodos era diferente. Tras estudiar el por qué estaba sucediendo esto, se llegó a la conclusión que al ser Kind un despliegue basado en contenedores, el servidor de métricas no estaba obteniendo datos de cada uno de los nodos, sino que los cogía directamente del *host* y esta era la razón por la cual el dato era correlativo y ascendente en cada una de las tomas.

Este aspecto se pudo finalmente confirmar tras ver en la documentación de *node-exporter* (del cual hablaremos en el apartado 3.3), que no es aconsejable lanzarlo en entorno contenerizado ya que las métricas son directamente obtenidas del sistema *host*. Llegados a este punto hubo que desechar la opción de crear el entorno con esta herramienta y buscar una nueva con la que desplegar nuestro clúster.

3.2.2 Segunda opción: Minikube

Tras ver que la opción basada en contenedores no era viable para los objetivos del proyecto, se buscó una alternativa basada en máquinas virtuales en la cual hubiese un aislamiento real de los recursos de los distintos nodos, aunque estos recursos fueran virtuales. Para ello y tras probar varias opciones, se optó por Minikube [16] con el controlador de VirtualBox, el cual cumple con los requisitos anteriormente expuestos.

Igual que en el caso anterior se ha habilitado un registro local donde guardar las imágenes de nuestras aplicaciones. El script de instalación de este clúster lo podemos ver en el Anexo A.3. El entorno se generó con 5 nodos virtuales, con 4 *CPU*s, 4 GB de *RAM* y 50 GB de disco para cada uno de ellos.

Una vez teníamos el entorno funcional, el siguiente paso sería ver cómo obtener la información de cada uno de los nodos.

3.3 SERVIDOR DE MÉTRICAS Y ANÁLISIS.

Para obtener las métricas del sistema, tales como uso de *CPU*, *RAM* o ancho de banda, hicimos uso de Prometheus [17], que junto a *node-exporter* [18], es la herramienta más extendida para este fin. Prometheus es un sistema de monitorización y alerta de código abierto desarrollado en 2012. Obtiene métricas de los sistemas monitorizados y los almacena como series temporales.

Además, es totalmente compatible con Kubernetes, pudiendo encontrarse múltiples manuales para desplegarlo de manera simple [19][20]. *Node-exporter* es un componente de

Prometheus que expone una basta cantidad de métricas *hardware* y del *kernel*. En el Anexo A.4 podemos ver como quedaron desplegados componentes.

Una vez el servidor de métricas está recopilando datos de los distintos nodos se procede a realizar un análisis entre el casi millar de métricas disponibles. Este estudio pretende dar luz sobre cuáles de estas métricas podrían ser útiles para usar con nuestro planificador, pero como veremos más adelante, el planificador realmente podrá consultar cualquiera de ellas.

Tras estudiar las métricas que aporta el componente *node-exporter* las hemos catalogado según su tipo en métricas de red, disco y *CPU*, A continuación, detallamos las métricas extraídas en nuestro estudio, y que a priori nos resultan de mayor interés para realizar la planificación de recursos.

3.3.1 Métricas de red

Las métricas de *node-exporter* [21] normalmente más usadas para obtener el uso del ancho de banda se obtienen del módulo *netdev*. Algunas de las más interesantes son *node_network_receive_bytes_total* y *node_network_transmit_bytes_total*, para conseguir el número de bytes recibidos y transmitidos respectivamente por una interfaz de red en un determinado nodo.

Existen otros contadores de métricas los cuales se obtienen de */proc/net/dev*, como pueden ser paquetes, errores, caídas, ... los cuales existen tanto para transmitir como para recibir. Otras, como las colisiones se obtienen para la transmisión únicamente. A partir de todas estas, una métrica interesante para usar en nuestro proyecto puede ser la ratio de errores de transmisión de datos (RETD) que podríamos obtener según la Ecuación 1, teniendo en cuenta que *node-exporter* obtiene los datos cada 5 ms

$$RETD = \frac{ratio(node_network_transmit_errs_total)}{ratio(node_network_transmit_packets_total)}$$

Ecuación 1. Ratio de errores de transmisión de datos

Por otro lado, la disponibilidad de los nodos está directamente relacionada con la capacidad de respuesta de los enlaces. En nuestro sistema, esta métrica se puede obtener a través de los valores obtenidos *node_network_carrier_up_changes_total* y *node_network_carrier_down_changes_total*.

3.3.2 Métricas de disco

La mayor parte de las métricas de disco se obtienen de la herramienta *iostat* [22], para el ancho de banda de acceso a disco lo más habitual es hacer uso de las métricas *node_disk_read_bytes_total* y *node_disk_written_bytes_total*.

También, podrían ser interesantes las métricas que nos indican el número total de lecturas y escrituras a un determinado disco completadas correctamente, *node_disk_reads_completed_total*, *node_disk_writes_completed_total* respectivamente.

Las Ecuación 2 y Ecuación 3 muestran el tiempo medio de lectura (TLD) y escritura (TED) en disco, respectivamente. Hemos de tener también en cuenta que estas métricas son obtenidas cada 5ms por *node-exporter*.

$$TLD = \frac{\text{node_disk_read_time_seconds_total}}{\text{node_disk_reads_completed_total}}$$

Ecuación 2. Tiempo medio de lectura en disco

$$TED = \frac{\text{node_disk_write_time_seconds_total}}{\text{node_disk_write_completed_total}}$$

Ecuación 3. Tiempo medio de escritura en disco

Por último, la métrica *node_disk_io_now* puede ser útil para monitorizar la actividad de E/S en un disco, detectar posibles cuellos de botella de E/S y optimizar el rendimiento del sistema en consecuencia. *node_disk_io_now* es una instantánea del número actual de operaciones de E/S y no proporciona información sobre la velocidad o el rendimiento de las operaciones de E/S.

3.3.3 Métricas de CPU

Hemos seleccionado la métrica *node_cpu_seconds_total* [23], que se obtiene de */proc/stat* y nos indica el tiempo empleado por cada CPU en cada uno de los modos en los que se puede ejecutar (ver Tabla 2).

Modo	Descripción
user	Tiempo gastado en <i>userland</i> .
system	Tiempo gastado en el <i>kernel</i> .
iowait	Tiempo gastado esperando por I/O.
idle	Tiempo en el que la CPU no está haciendo nada.
irq&softirq	Tiempo sirviendo interrupción.
guest	Si estás ejecutando máquinas virtuales, la CPU que usan.
steal	Si estás usando una máquina virtual, tiempo que otras máquinas virtuales "roban"

Tabla 2: Modos de ejecución de la CPU

Estos modos son mutuamente excluyentes. Esta métrica puede ser útil para realizar un seguimiento del uso de la CPU en un nodo y obtener información sobre cómo se están utilizando los recursos del sistema. Por ejemplo, un alto valor de *iowait* indica un alto uso de CPU en entrada/salida; un alto valor de *user* o *system*, que la CPU está activa con un uso alto. Podremos sumar estos valores para obtener el valor medio de uso de todas las CPUs (VMU) de un servidor, como se muestra en la Ecuación 4.

$$VMU = \sum_{mode=0}^{m-1} \sum_{instance=0}^{i-1} ratio(node_cpu_seconds_total)$$

Ecuación 4. Valor medio de las CPUs de un servidor

3.4 PERSISTENCIA DE DATOS

Se van a realizar estudios de series temporales sobre las métricas extraídas a través de Prometheus. Por tanto, debemos persistir esos valores a lo largo del tiempo, ya que, si únicamente consultásemos la información en el momento de realizar la planificación, obtendríamos valores instantáneos los cuales no nos aportan la fotografía del sistema que necesitamos.

Al igual que en la creación del entorno local, el modo en el cual se ha decidido persistir los datos ha cambiado desde la idea inicial. En un principio se pensó en una base de datos Postgresql y un *adapter* que leyese los datos de Prometheus para insertarlos en ésta, sin embargo, se le vieron problemas de ejecución en un entorno tan pequeño como el disponible. Por tanto, se desplegó otra solución basada en una base de datos TimescaleDB y un componente de la misma compañía llamado Promscale y que sería el encargado de persistir la información de Prometheus. A continuación, se detallará como se realizó el diseño de ambas opciones y por qué decidimos quedarnos con la segunda.

3.4.1 Postgresql - *Custom adapter*

La primera opción que se barajó para realizar la persistencia de las métricas de Prometheus fue desplegar una base de datos Postgresql dentro de nuestro entorno local, para ello, se usó uno de los múltiples tutoriales que existen [24]. Además, se modificó el adaptador de Crunchy Data [25] para adaptarlo a las necesidades del proyecto. Este adaptador, sería el encargado de leer el API de Prometheus y de persistir esa información en la base de datos. A continuación, se describen brevemente las acciones que se realizaron para conseguir la adaptación. Una descripción detallada puede verse en el Anexo A.5.

- Se desplegó el adaptador dentro de un contenedor de Docker y se expusieron todos los parámetros de configuración para poder lanzarlo como un objeto *Deployment* de Kubernetes.
- Se creó un objeto *Service* para que el adaptador tuviera acceso a Prometheus y viceversa.
- Se realizó una modificación en el código del adaptador para crear vistas en la base de datos, con el objetivo de facilitar el acceso a los datos requeridos desde el planificador. Estas vistas fueron definidas durante el análisis.

Sin embargo, una vez montado todo el sistema de persistencia, se observó que este consumía muchos recursos, sobre todo el *adapter*, al cual se le fueron aumentando los recursos, pero aun así se reiniciaba asiduamente. Ya que el clúster que se ha usado para realizar el proyecto es local y posee pocos recursos, se decidió buscar una alternativa.

3.4.2 Timescaledb - Promscale

Tras estudiar más en profundidad las opciones disponibles para realizar la persistencia de las métricas, se encontró el producto TimescaleDB [26], una base de datos basada en *Postgresql*, muy potente y que ofrece herramientas y funciones para realizar estudios estadísticos de series temporales, algo que será útil para realizar los cálculos de nuestro planificador.

Junto a ella, también encontramos una herramienta desarrollada por la misma empresa llamada Promscale [27]. Se trata de un adaptador que persiste cada determinado tiempo la información existente en Prometheus en la base de datos. En la Figura 5 podemos ver la arquitectura genérica de esta solución.

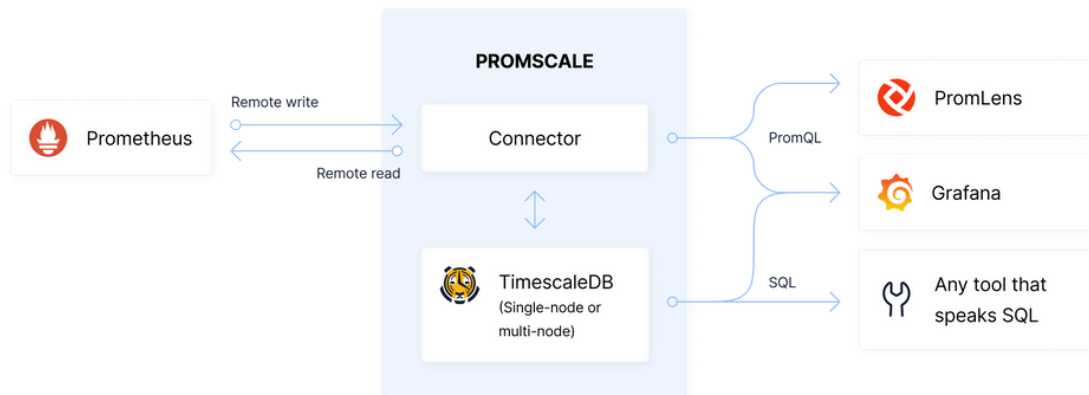


Figura 5. Arquitectura de la solución de persistencia de datos [28]

Tras realizar múltiples pruebas con ella se pudo confirmar como es una opción adecuada para los requisitos que teníamos para el desarrollo del planificador. Además, consume muchos menos recursos que la alternativa anterior y es más estable. Por otro lado, parece que hay mucho más trabajo y contribuidores detrás de ella.

La instalación de ambas herramientas se ha realizado mediante Helm [29], que es un administrador de paquetes de Kubernetes que facilita la instalación y actualización de aplicaciones complejas. Los *scripts* de instalación pueden verse en el Anexo A.6 y en ellos se puede ver que es necesario crear la base de datos *tsdb* antes de instalar Promscale.

La herramienta crea una vista por cada una de las métricas de Prometheus y periódicamente va introduciendo datos en ella. La estructura de estas vistas es similar en todas ellas. En la Figura 6, vemos la estructura de la vista que representa la métrica *node_network_receive_bytes_total*.

Column	Type
time	timestamp with time zone
value	double precision
series_id	bigint
labels	label_array
app_id	integer
controller_revision_hash_id	integer
device_id	integer
instance_id	integer
job_id	integer
kubernetes_namespace_id	integer
kubernetes_pod_name_id	integer
pod_template_generation_id	integer

Figura 6. Estructura de la vista *node_network_receive_bytes_total*

De los cuales nos interesan, **time** que representa el *timestamp* de la toma, **value** el valor de la toma, **instance_id** el id de la instancia que haría referencia al nodo de Kubernetes y **device_id** como el id del dispositivo, en este caso hace referencia a la interfaz de red.

Con estos datos, debemos tener en cuenta que para implementar nuevos algoritmos de planificación que consideren estas métricas siempre necesitaremos:

- El campo *instance_id*, ya que deberemos agrupar los datos por este campo para posteriormente asignar las prioridades
- El campo *time*, ya que siempre haremos estudios sobre series temporales.
- El campo *value*, será con el que obtengamos el valor del estudio
- Luego habrá otros campos, como en este caso *device_id*, que dependerán de la métrica que deberemos filtrar para obtener la información que necesitamos. En este ejemplo, filtraremos por la interfaz que usa Minikube para transmitir datos.

Con todo esto, podemos obtener los 10 últimos valores de la vista *node_network_receive_bytes_total* para el interfaz *eth1*. Ver la consulta en el Código 1. Consulta de *node_network_receive_bytes_total* con detalle. y el resultado obtenido en la Figura 7.

```

SELECT time,
       value,
       val(device_id) AS device,
       val(instance_id) AS instance
FROM node_network_receive_bytes_total
WHERE val(device_id) = 'eth0'
ORDER BY time
LIMIT 10;

```

Código 1. Consulta de *node_network_receive_bytes_total* con detalle.

time	value	device	instance
2023-06-08 21:05:49.314+00	659727830	eth0	192.168.59.244:9100
2023-06-08 21:05:49.439+00	562883968	eth0	192.168.59.246:9100
2023-06-08 21:05:50.577+00	563042213	eth0	192.168.59.242:9100
2023-06-08 21:05:51.476+00	562883968	eth0	192.168.59.246:9100
2023-06-08 21:05:51.66+00	572540988	eth0	192.168.59.245:9100
2023-06-08 21:05:51.825+00	563042293	eth0	192.168.59.242:9100
2023-06-08 21:05:51.827+00	572540988	eth0	192.168.59.245:9100
2023-06-08 21:05:52.971+00	659727830	eth0	192.168.59.244:9100
2023-06-08 21:05:54.314+00	659727830	eth0	192.168.59.244:9100
2023-06-08 21:05:54.439+00	562883968	eth0	192.168.59.246:9100

Figura 7. Resultados de *node_network_receive_bytes_total* con detalle.

Como hemos comentado, anteriormente este formato de datos no sería útil para lo que necesitamos en el planificador, que debería agrupar los datos por instancia. Agrupar estos datos y operar sobre ellos nos permite realizar diversos cálculos estadísticos como por ejemplo sumas, valores máximos, mínimos, medias, ... y otras operaciones analíticas más complejas que aporta TimescaleDB [30]. Por ejemplo, en la Figura 8, podemos ver el resultado de obtener el valor máximo de la métrica *node_network_receive_bytes_total* para cada uno de los nodos con la consulta definida en el Código 2. Consulta del valor máximo de la métrica *node_network_receive_bytes_total* por nodo.

```

SELECT val(instance_id) instance,
       max(value) AS value
FROM node_network_receive_bytes_total
WHERE val(device_id) = 'eth0'
GROUP BY val(instance_id)

```

Código 2. Consulta del valor máximo de la métrica *node_network_receive_bytes_total* por nodo.

instance	value
192.168.59.242:9100	563045532
192.168.59.243:9100	1341951015
192.168.59.244:9100	1011889392
192.168.59.245:9100	1058163607
192.168.59.246:9100	915048842

Figura 8. Valor máximo de la métrica *node_network_receive_bytes_total* por nodo

Con esta información obtenida, podríamos asignar prioridades a cada uno de los nodos según las necesidades del usuario.

3.5 CREACIÓN DEL PLANIFICADOR

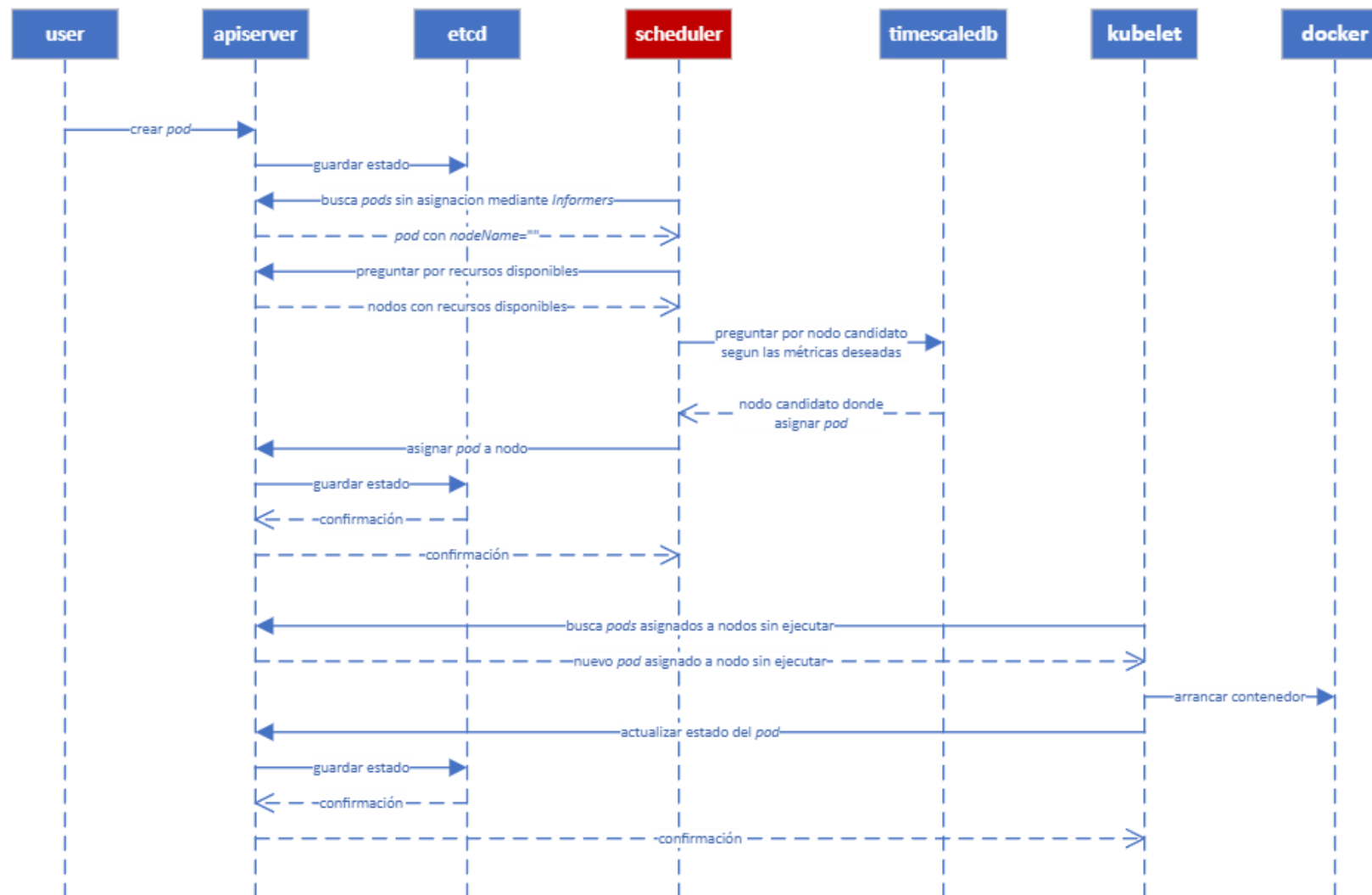
La creación del planificador es uno de los puntos más importantes de este trabajo. Para entender cómo debía realizarse se leyó la bibliografía existente, tanto de Kubernetes como de ejemplos de planificadores aportados por la comunidad [31].

Uno de ellos [32], sirvió especialmente para entender cómo funciona el proceso de planificación y valió como base para desarrollar el nuestro propio. Es un planificador con una política aleatoria de asignación de nodos a *pods* y se usará también en las pruebas del siguiente apartado. Para entender el funcionamiento del planificador, vamos a explicar cada una de sus partes por separado.

En la Figura 9 podemos ver un diagrama de secuencia donde se puede ver como se crea un *pod* y la interacción entre el planificador y el resto de los componentes. La secuencia de acciones que se llevan a cabo serían las siguientes:

1. El usuario crea un *pod*, petición que llega a Kubernetes a través del *API server*. Esta información se guarda en *etcd*.
2. El planificador está continuamente escuchando sobre la creación de nuevos *pods*, esta información le llega a través de un *informer*.
3. Cuando el planificador recibe un nuevo *pod*, pregunta al *API Server* por los nodos que tienen capacidad suficiente para alojar el *pod*. Éste le devuelve la lista de candidatos.
4. El planificador consultará a la base de datos, cual de entre los nodos candidatos es el que más se ajusta a los requisitos de este planificador. TimescaleDB devolverá el mejor nodo.
5. El planificador asignará el nombre del nodo candidato al *pod* y avisará al *API server* de ello. Éste guardará la información en *etcd*.
6. El *kubelet* de ese nodo, que continuamente está escuchando por *pods* con nodo asignado, pero sin ejecutar, lo seleccionará y arrancará el contenedor.
7. Por último, el *kubelet* informará al *API server* del cambio de estado, y éste a su vez guardará la información en *etcd*.

Algunos de estos conceptos no han sido detallados hasta ahora, pero se profundizará en todos ellos a lo largo del capítulo.

Figura 9. Diagrama de secuencia de creación de un *pod*

3.5.1 Descripción general

El planificador es una aplicación escrita en Golang que será gestionada por un objeto *Deployment* de Kubernetes, proveyéndole de tolerancia a fallos en caso de error.

Golang o Go [33] es un lenguaje de programación concurrente y tipado estático inspirado en la sintaxis de C, desarrollado por Google. Muchas aplicaciones actuales son desarrolladas en este lenguaje, como por ejemplo Kubernetes. Será usado para la realización de las distintas partes del proyecto.

Al crearse la aplicación, se creará una cola donde se incluirán los *Pods* pendientes de planificar. También habrá dos *Informers*, que son unos componentes de Kubernetes que se encargan de informar a los *controllers* de cambios en determinados objetos, y de los que hablaremos en profundidad más adelante. Uno recogerá los cambios en los nodos del clúster y otro estará en escucha de nuevos *Pods* que deban ser planificados por el planificador, introduciéndolos en la cola según lleguen.

Por otro lado, habrá un proceso de planificación controlando esa cola para buscar el nodo que más se adapta a las condiciones indicadas por el administrador, asignando prioridades a cada uno de los nodos, para posteriormente asignar el *Pod* al nodo más prioritario. Se puede consultar el código del planificador en el CD adjunto a esta memoria.

3.5.2 Parámetros de entrada

La aplicación recibe los parámetros de entrada que se muestran en la Tabla 3, los cuales serán parametrizables por el usuario al lanzar el planificador con el operador.

Sección	Parámetro	Descripción
TimescaleDB	host	Url de conexión a la base de datos.
	port	Puerto de conexión a la base de datos.
	user	Usuario de conexión a la base de datos.
	password	Contraseña del usuario de conexión a la base de datos.
	database	Nombre de la base de datos.
	authentication	Modo de autenticación de la base de datos.

Metrics	name	Nombre de la métrica y a la vez de la vista de la base de datos.
	startDate	Fecha inicio de la consulta.
	endDate	Fecha fin de la consulta.
	operation	Operación que se realizara sobre los campos values de la consulta.
	priorityOrder	Orden en el que se asignaran las prioridades: asc o desc.
	isSecondLevel	Indica si la consulta es de uno o dos niveles. Esto hace referencia al número de agrupaciones de datos que realizara la consulta.
	secondLevelSelect	Campos que se incluirán en el segundo nivel de la consulta.
	secondLevelGroup	Campos por los que se agrupara en el segundo nivel de la consulta.
	filter	Lista de condiciones por las que se filtrara la consulta.

Tabla 3. Parámetros de entrada del planificador

3.5.3 Informers

Dentro de nuestro planificador debemos estar continuamente informados del estado de los nodos del clúster, así como de la creación de nuevos *pods* que planificar. Si estuviésemos preguntando periódicamente al API server sobre esta información podríamos degradar el rendimiento del clúster.

En Kubernetes, los *informers* son componentes clave que permiten a los *controllers* y otros sistemas de Kubernetes recibir actualizaciones sobre los recursos y eventos en el clúster. Un *informer* es una interfaz que se utiliza para acceder a los objetos de Kubernetes almacenados en *etcd*.

Cuando se crea un *informer*, se especifica un tipo de objeto de Kubernetes que se desea observar. El *informer* monitoriza continuamente los cambios en los objetos de ese tipo y notifica a los *controllers* y otros componentes interesados sobre las actualizaciones. Los *informers* son una parte esencial de la arquitectura de controlador de Kubernetes y se utilizan para construir *controllers* personalizados y sistemas de administración basados en eventos.

Los *informers* ofrecen una manera eficiente de rastrear y reaccionar a los cambios en los recursos de Kubernetes. Al utilizar *informers*, los *controllers* y otros sistemas pueden mantener un estado actualizado de los recursos y tomar acciones en función de los eventos ocurridos en el clúster. Esto facilita la automatización y gestión de aplicaciones en entornos Kubernetes.

Para integrar los *informers* en un *controller* de Kubernetes, es necesario importar los paquetes adecuados y deberemos crear un cliente para interactuar con el clúster. A continuación, se configura el *informer* para el tipo de objeto que se desea observar, y se definen los controladores de eventos que se activarán al producirse cambios en los recursos. Después de iniciar el *informer*, el *controller* puede procesar los eventos recibidos, actualizar su estado interno y tomar las acciones necesarias en respuesta a los cambios en el clúster. De esta manera, los *informers* desempeñan un papel fundamental en la sincronización y la toma de decisiones dentro de los controladores de Kubernetes.

Nuestro planificador levantará dos *informers*, uno para sondear los cambios de los nodos y otro para los *Pods*. El *informer* de *Pods* solo recogerá los objetos que deban ser planificados por nuestro planificador y para ello comparará el nombre de nuestro planificador con el del campo del *Pod spec.SchedulerName*. En caso de coincidir lo incluirá en la cola de *Pods* pendientes de planificar.

3.5.4 Filtrado por recursos

Una vez se recibe un evento indicando que un nuevo *Pod* debe ser planificado, lo primero que hará el planificador será obtener el estado de cada uno de los nodos. Con esa información se compararán la suma de los requisitos de *CPU* y memoria de cada uno de los contenedores del *Pod* con la capacidad disponible de cada uno de los nodos, filtrándose los que no tienen recursos suficientes. La función mostrada en el Código 3 se aplicará a cada uno de los nodos.

Además de este filtrado, se puede realizar un filtro previo en el que el usuario puede decidir si no le interesa alguno de los nodos disponibles. Por ejemplo, el usuario podría filtrar el nodo donde se encuentra la solución de monitorización y el *control plane*.

3.5.5 Filtrado por métricas mediante consultas a TimescaleDB

Una vez filtrados los nodos con capacidad para ser asignados, el planificador construirá la consulta que se realizará sobre la base de datos para obtener las prioridades según los

parámetros indicados por el usuario. Existen dos tipos de consultas, las que hemos llamado de un nivel y las de dos niveles. Para entender cómo se construye cada una de ellas lo mejor es explicarlo mediante ejemplos.

```
func (s *Scheduler) fitResourcesPredicate(node *v1.Node, pod *v1.Pod) bool {

    var podCpu resource.Quantity
    var podMemory resource.Quantity
    var nodeCpu resource.Quantity
    var nodeMem resource.Quantity

    for _, container := range pod.Spec.Containers {
        podCpu.Add(*container.Resources.Requests.Cpu())
        podMemory.Add(*container.Resources.Requests.Memory())
    }

    pods, _ := s.clientset.CoreV1().Pods("").List(context.TODO(), metav1.ListOptions{
        FieldSelector: "spec.nodeName=" + node.Name,
    })

    for _, npod := range pods.Items {
        for _, ncontainer := range npod.Spec.Containers {
            nodeCpu.Add(*ncontainer.Resources.Requests.Cpu())
            nodeMem.Add(*ncontainer.Resources.Requests.Memory())
        }
    }
    freeCpu := node.Status.Allocatable.Cpu()
    freeCpu.Sub(nodeCpu)
    freeMem := node.Status.Allocatable.Memory()
    freeMem.Sub(nodeMem)

    if freeCpu.Cmp(podCpu) == -1 || freeMem.Cmp(podMemory) == -1 {
        return false
    }
    return true
}
```

Código 3. Función de filtrado por recursos

3.5.5.1 Consulta de un nivel.

Queremos estudiar cual ha sido el incremento de transmisión de datos en cada uno de los nodos en los últimos 30 minutos en la interfaz de red *eth1* y que sea ordenado de manera descendente. Para ello usaremos la consulta definida en el Código 4.

```
SELECT Val(instance_id) node,
       last(value,time) - first(value,time) value
FROM   node_network_transmit_bytes_total
WHERE  time >=now()- interval '30 MINUTES'
AND    time <=now()
AND    val(device_id) IN ('eth1')
GROUP BY val(instance_id)
ORDER BY last(value,time) - first(value,time) DESC
```

Código 4. Consulta de un nivel

La operación *last* y *first* obtiene el ultimo y el primer valor de la serie temporal en referencia al tiempo. Restándolos conseguimos el incremento. En la Tabla 4 podemos ver los parámetros que debemos indicar para generar esta consulta.

Parámetro	Valor
name	node_network_transmit_bytes_total
startDate	now()- interval '30 MINUTES'
endDate	now().
operation	last(value,time) - first(value,time)
priorityOrder	desc
isSecondLevel	false
filter	val(device_id) IN ('eth1')

Tabla 4. Parámetros para consulta de un nivel.

El planificador encapsula la consulta anterior quedando tal como se ve en el Código 5. Consulta de un nivel generada por el planificador.

```
SELECT row_number() OVER() rowid,
       a.node,
       a.value
FROM   (
        SELECT left(Val(instance_id),-5) node,
               Last(value,time) - first(value,time) value
        FROM   node_network_transmit_bytes_total
        WHERE  value <> 'Nan'
              AND time >=Now()- interval '5 MINUTES'
              AND time <=now()
              AND val(device_id) IN ('eth1')
        GROUP BY val(instance_id)
        ORDER BY Last(value,time) - first(value,time) DESC) AS a;
```

Código 5. Consulta de un nivel generada por el planificador

Esto se hace para asignar las prioridades de manera ascendente a cada uno de los registros. Además, elimina el puerto del campo instancia. El resultado final se puede ver en la Figura 10.

rowid	node	value
1	192.168.59.249	498481376
2	192.168.59.251	166293788
3	192.168.59.250	146738482
4	192.168.59.247	31502343
5	192.168.59.248	5103393

Figura 10. Resultado de consulta de un nivel.

3.5.5.2 Consulta de dos niveles

Existen otro tipo de consultas más complejas las cuales no podríamos resolver con el modelo anterior. Por ejemplo, imaginemos un escenario, en el que queremos priorizar la asignación en función del nodo que ha tenido sus *CPUs* más tiempo en modo *idle* (en espera), calculando el incremento de tiempo en los últimos 15 minutos. La consulta anterior no nos valdría porque por cada nodo existen *X CPUs* y para calcular el tiempo total, necesitaríamos sumar el tiempo de cada una de ellas. La consulta se puede ver en el Código 6.

```
SELECT  a.node,
Sum(a.value) value
FROM    (
  SELECT  Val(instance_id)      node,
  val(cpu_id) cpu,
  last(value,time) - first(value,time) value
  FROM    node_cpu_seconds_total
  WHERE    time >=Now() - interval '15 MINUTE'
  AND      time <=now()
  AND      val(mode_id)='idle'
  GROUP BY val(instance_id),
  val(cpu_id)) a
GROUP BY node
ORDER BY sum(value) ASC
```

Código 6. Consulta de dos niveles.

En la Tabla 5. Parámetros para consulta de dos niveles. podemos ver los parámetros necesarios para generar esta consulta.

Parámetro	Valor
name	node_cpu_seconds_total
startDate	now()- interval '15 MINUTES'
endDate	now().
operation	sum
priorityOrder	asc
isSecondLevel	true
filter	filter: val(mode_id)='idle'
secondLevelSelect	"val(instance_id)node", "val(cpu_id)cpu", "last(value;time)-first(value;time)value"]
SecondLevelGroup	["val(instance_id)", "val(cpu_id)"]

Tabla 5. Parámetros para consulta de dos niveles.

Notemos que los campos *filter*, *secondLevelSelect* y *secondLevelGroup* son arrays de cadenas de caracteres. El planificador encapsula la consulta anterior y añade una serie de campos y filtros, quedando la consulta final como se puede ver en el Código 7. Consulta de

dos niveles generada por el planificador. El formato del resultado de esta consulta es igual al anterior.

```
SELECT row_number() OVER() rowid,
       b.node,
       b.value
FROM   (
        SELECT left(a.node,-5) node,
               Sum(a.value)      value
        FROM   (
                SELECT Val(instance_id)      node,
                       val(cpu_id) cpu,
                       last(value,time) - first(value,time) value
                FROM   node_cpu_seconds_total
                WHERE  value <> 'Nan'
                AND    time >=Now() - interval '15 MINUTE'
                AND    time <=now()
                AND    val(mode_id)='idle'
                GROUP BY val(instance_id),
                       val(cpu_id) ) a
        GROUP BY node
        ORDER BY sum(value) ASC) AS b;
```

Código 7. Consulta de dos niveles generada por el planificador

3.5.6 Asignación de prioridades y elección de nodo

Una vez realizado el filtrado inicial y asignadas las prioridades a cada uno de los nodos seleccionables, únicamente quedará asignar el *pod* al nodo. Esto se realizará llamando al API de Kubernetes, más concretamente al método *Bind* del objeto *pod*. Por último, se generará un evento informando del resultado de la planificación.

3.6 CREACIÓN DEL OPERADOR DE PLANIFICADORES

Como último apartado del capítulo, vamos a ver como se ha desarrollado el operador encargado de lanzar los planificadores que a su vez lanzarán los *pods* con las condiciones que nosotros previamente indiquemos. En este caso también lo vamos a separar en distintos subapartados en los que hablaremos de:

- **Kubebuilder y scaffolding:** comentaremos cómo hemos creado el esqueleto o *scaffolding* de nuestro operador con las herramientas que nos proporciona Kubebuilder.

- **Definición del *Cusom Resource*:** describiremos cómo es el objeto que hemos creado para representar nuestro planificador, los campos que lo forman y las opciones de parametrización que dispone.
- ***Controller* y objetos:** por último, explicaremos cómo funciona el *controller* que se encarga de gestionar el ciclo de vida de nuestro planificador y los objetos de Kubernetes que forman nuestro *Custom Resource*.

3.6.1 Kubebuilder y scaffolding

Kubebuilder [34] es un framework desarrollado en *Golang* para la creación de APIs de Kubernetes usando *Custom Resource Definitions* (CRDs) [35]. Está desarrollado por encima de las librerías *controler-runtime* y *controller-tools* [36]. En este proyecto ha sido usado para facilitar el desarrollo del operador que gestionará los planificadores.

En Kubernetes un *CRD* es una característica que permite a los usuarios definir y utilizar recursos personalizados en el clúster. Es una extensión del modelo de objetos de Kubernetes para admitir recursos que no están disponibles por defecto.

Un *CRD* define una estructura de datos personalizada que representa un nuevo tipo de recurso en Kubernetes. Al crear un *CRD*, los usuarios pueden definir los campos, las propiedades y los comportamientos específicos de su recurso personalizado. Una vez que el *CRD* se ha creado y registrado en el clúster, los usuarios pueden crear, actualizar y eliminar instancias de ese recurso personalizado utilizando los comandos y las API de Kubernetes.

Un *Custom Resource (CR)* [37] es una extensión del API de Kubernetes que no tiene por qué estar disponible en la instalación por defecto de Kubernetes. Son los recursos que se generan a partir de la definición del *CRD*.

Kubebuilder es la herramienta que se ha usado para generar el operador y crear el API del objeto *MetricScheduler* usando *CRD*. También se crearán los *webhooks* que servirán para asignar valores por defecto y validar los *CRs* creados. El *script* se puede ver en el Código 8.


```

go mod init scheduler-operator
kubebuilder init --domain uclm.es
kubebuilder create api --group scheduler --kind MetricScheduler --version v1 --namespaced
false
kubebuilder create webhook --group scheduler --version v1 --kind MetricScheduler --
defaulting --programmatic-validation

```

Código 8. Script de creación del API de MetricScheduler

Esto creará el esqueleto del operador y facilita el desarrollo de este. Aun así, deberemos definir el fichero *types* donde se incluirán los parámetros que tendrá nuestro *CR*, el *controller* que será el encargado de gestionar el ciclo de vida y los *webhooks* para realizar el *defaulting* y la validación de la creación, actualización y borrado de los objetos.

3.6.2 Definición del *Custom Resource*

Para la creación del nuevo *Custom Resource MetricScheduler* hemos definido en él, además de los parámetros que necesita el planificador, una serie de parámetros y características de los objetos de Kubernetes que forman parte de él y que están detallados en la Tabla 6, por ejemplo, los recursos con los que se levantara el *pod*, la imagen de Docker del planificador, ...

Parámetro	Descripción
image	Imagen de Docker del planificador.
imagePullPolicy	Política de descarga de la imagen de Docker.
instance	Numero de instancias del planificador.
resources.request.cpu	Cantidad de <i>CPU</i> con la que se levantara el <i>pod</i> .
resources.request.memory	Cantidad de memoria con la que se levantara el <i>pod</i> .
limit.request.cpu	Límite de <i>CPU</i> con la que se puede levantar el <i>pod</i> .
limit.request.memory	Límite de memoria con la que se puede levantar el <i>pod</i> .

Tabla 6. Parámetros comunes de despliegue del planificador.

Por otra parte, tendremos una sección con la información de conexión a *TimescaleDB* y otra definiendo las características de la métrica con la que el planificador decidirá como planificar los *pods*.

Un ejemplo del descriptor que usaremos para desplegar nuestro planificador se puede ver en el Código 9. En él, podemos ver que es un objeto de tipo (*Kind*) *MetricScheduler*, que tiene como metadatos el nombre (*name*) con el que se desplegará y el *namespace* (Kubernetes

soporta múltiples clústeres virtuales respaldados por el mismo clúster físico. Estos clústeres virtuales se denominan espacios de nombres o *namespaces*) donde se desplegará. Además, en la sección *spec* vemos todas los parámetros y subsecciones de los que hemos hablado anteriormente.

```
apiVersion: scheduler.uclm.es/v1
kind: MetricScheduler
metadata:
  name: metricscheduler
  namespace: ns1
spec:
  image: localhost:5000/albertogomez/scheduler:0.0.0
  imagePullPolicy: Always
  instances: 1
  resources:
    requests:
      cpu: "1"
      memory: "1024Mi"
    limits:
      cpu: "1"
      memory: "1024Mi"
  timescaledb:
    host: timescaledb.monitoring
    port: "5432"
    user: postgres
    password: patroni
    database: tsdb
    authenticationType: MD5
  metric:
    name: "node_cpu_seconds_total"
    startDate: "now()- INTERVAL '10 MINUTE'"
    endDate: "now()"
    operation: "sum"
    priorityOrder: "asc"
    isSecondLevel: true
    secondLevelSelect:
      - "val(instance_id)node"
      - "val(cpu_id)cpu"
      - "Last(value;time)-first(value;time) value"
    secondLevelGroup:
      - "val(instance_id)"
      - "val(cpu_id)"
    filters:
      - "val(mode_id)='idle'"
```

Código 9. Descriptor CR MetricScheduler

3.6.3 *Controller* y objetos.

La parte más importante de nuestro operador es el *controller*. En él, se define la función *Renconcile*, que implementa el *reconciliador* de nuestro operador. Este elemento se encargará de comprobar cada X tiempo que todos los objetos de nuestro *CR* se encuentran ejecutándose correctamente. En caso de que alguno de ellos no esté disponible debido a un fallo puntual o porque haya sido eliminado de forma fortuita o malintencionada por un

usuario o servicio, el *controller* se comunicará con el *API server* para llevar a cabo las tareas necesarias para volver a crearlo.

El periodo de reconciliación será diferente si se ha producido un error en la creación de los objetos o no. El *controller* también se encargará de eliminar ordenadamente un *CR*. Y aunque no es nuestro caso ya que estamos trabajando en una prueba de concepto, desde el *controller* se harán las llamadas para realizar la actualización del estado del *CR*. Se puede consultar el código del *controller* y de la creación de los objetos creados por el operador en el CD adjunto a esta memoria.

El control de acceso en Kubernetes se realiza mediante *RBAC* (*Role-based access control*). Para ello se asignará a la identidad de nuestro servicio objetos *Role* o *ClusterRole* los cuales definen los permisos sobre los que se tendrá acceso. La diferencia entre *Role* y *ClusterRole* es que los primeros hacen referencia a permisos sobre un determinado *Namespace* y los segundos a permisos sobre el clúster completo.

En esta primera iteración del operador se han creado los siguientes objetos:

- ***Service Account***: Sirve para proveer al planificador de una identidad dentro de un clúster de Kubernetes. A este *Service Account* se le asignaran las políticas de seguridad que deseemos que nuestro objeto disponga. Además, de proveerle de un sistema de autenticación.
- ***Cluster Role Binding***: Sirve para asociar un determinado *Cluster Role* al *Service Account* antes definido. En este caso, el planificador necesita disponer del *Cluster Role system:kube-scheduler* para poder planificar los *Pods*.
- ***Deployment***: Provee de actualizaciones declarativas a un *pod*. Mediante el *deployment* se describe el estado deseado y el *controller* se encarga de modificar el estado actual al deseado. En este caso sirve para lanzar y actualizar el *pod* del planificador.

CAPÍTULO 4. RESULTADOS

En este capítulo vamos a comprobar el funcionamiento e interacción de los distintos componentes diseñados en nuestro planificador y realizaremos una evaluación comparativa frente a otros planificadores aplicando una prueba de carga.

4.1 PRUEBA FUNCIONAL

En este primer apartado, vamos a desplegar el sistema al completo con todos sus módulos y comprobaremos que funcionan correctamente e interactúan como deben entre ellos. Una vez lanzado el *script* completo de despliegue detallado en los Anexos A.3, A.4 y A.6, podemos ver en la Figura 11. Entorno desplegado que se ha generado el entorno correctamente.

NAME	STATUS	ROLES	AGE	VERSION
tfm-dev	Ready	control-plane	11m	v1.26.3
tfm-dev-m02	Ready	<none>	10m	v1.26.3
tfm-dev-m03	Ready	<none>	9m26s	v1.26.3
tfm-dev-m04	Ready	<none>	8m10s	v1.26.3
tfm-dev-m05	Ready	<none>	7m7s	v1.26.3

Figura 11. Entorno desplegado

Además de los *namespaces* que se generan por defecto en Kubernetes, hemos creado un *namespace* llamado *monitoring* donde se instalarán todos los módulos de monitorización. En él, se han instalado:

- Servidor de Prometheus, al cual las instancias de *node-exporter* irán mandando las métricas.
- *Node-exporter*, un *pod* en cada uno de los nodos para poder monitorizarlos.
- TimescaleDB: donde se registrarán las métricas periódicamente.
- Promscale: que se encargara de leer las métricas de Prometheus y persistirlas en TimescaleDB.

En la Figura 12 podemos ver los *pods* desplegados en este *namespace* y que forman todo el *stack* de monitorización. Se ha decidido instalar todos estos componentes en el nodo *tfn-dev-m02*, para no sobrecargar el resto de los nodos con estas tareas la hora de realizar las evaluaciones.

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
node-exporter-bn7qs	1/1	Running	0	46m	192.168.59.253	tfn-dev-m02
node-exporter-cmhk2	1/1	Running	0	46m	192.168.59.254	tfn-dev
node-exporter-h2cc7	1/1	Running	0	46m	192.168.59.250	tfn-dev-m05
node-exporter-r577w	1/1	Running	0	46m	192.168.59.251	tfn-dev-m04
node-exporter-x2qtp	1/1	Running	0	46m	192.168.59.252	tfn-dev-m03
prometheus-deployment-84d46d5bcb-7dvfn	1/1	Running	0	46m	10.244.1.3	tfn-dev-m02
promscale-55688b59d7-qc6r7	1/1	Running	0	44m	10.244.1.5	tfn-dev-m02
timescaledb-0	1/1	Running	0	46m	10.244.1.4	tfn-dev-m02

Figura 12. *Pods* del *namespace monitoring*.

Además, podemos comprobar en los logs como Promscale ingesta correctamente las métricas en TimescaleDB (Figura 13).

```
level=info ts=2023-06-25T18:27:01.819Z caller=throughput.go:90 msg="ingestor throughput" samples/sec=44000 metrics-max-sent-ts=2023-06-25T18:24:26.367Z
level=info ts=2023-06-25T18:27:03.817Z caller=throughput.go:90 msg="ingestor throughput" samples/sec=22000 metrics-max-sent-ts=2023-06-25T18:24:37.65Z
level=info ts=2023-06-25T18:27:04.818Z caller=throughput.go:90 msg="ingestor throughput" samples/sec=22000 metrics-max-sent-ts=2023-06-25T18:24:37.65Z
level=info ts=2023-06-25T18:27:06.817Z caller=throughput.go:90 msg="ingestor throughput" samples/sec=44000 metrics-max-sent-ts=2023-06-25T18:24:37.65Z
level=info ts=2023-06-25T18:27:07.818Z caller=throughput.go:90 msg="ingestor throughput" samples/sec=22000 metrics-max-sent-ts=2023-06-25T18:24:37.65Z
```

Figura 13. Logs de *Promscale* ingestando métricas.

Por otro lado, hemos creado un *namespace* *ns1*, donde hemos desplegado el operador que gestionará los planificadores y un planificador de prueba para validar el entorno. En la Figura 14 podemos ver el operador ejecutándose correctamente en espera de nuevas peticiones.

```

1.68891645963889e+09 INFO      setup      Managing for Namespaces ns1
1.6889164597335262e+09 INFO      controller-runtime.metrics Metrics server is starting to listen {"addr": ":8080"}
1.6889164597338283e+09 INFO      setup      starting manager
1.6889164597343767e+09 INFO      Starting server {"path": "/metrics", "kind": "metrics", "addr": "[::]:8080"}
10709 15:27:39.734565 7 leaderelection.go:248] attempting to acquire leader lease ns1/6157578d.ucln.es...
1.6889164597349703e+09 INFO      Starting server {"kind": "health probe", "addr": "[::]:8081"}
10709 15:27:39.746961 7 leaderelection.go:258] successfully acquired lease ns1/6157578d.ucln.es
1.6889164597473999e+09 INFO      Starting EventSource {"controller": "metricscheduler", "controllerGroup": "scheduler.ucln.es", "controllerKind": "MetricScheduler", "source": {"kind source": "v1.MetricScheduler"}}
1.6889164597472734e+09 DEBUG     events manager_d2eab428-1453-43cf-a593-6a5156d15500 became leader {"type": "Normal", "object": {"kind": "Lease", "namespace": "ns1", "name": "6157578d.ucln.es", "uid": "87cd563e-d2c3-401a-91a5-a892177362d4", "apiVersion": "coordination.k8s.io/v1", "resourceVersion": "3161"}, "reason": "LeaderElection"}}
1.6889164597476459e+09 INFO      Starting Controller {"controller": "metricscheduler", "controllerGroup": "scheduler.ucln.es", "controllerKind": "MetricScheduler"}
1.6889164598492012e+09 INFO      Starting workers {"controller": "metricscheduler", "controllerGroup": "scheduler.ucln.es", "controllerKind": "MetricScheduler", "worker count": 1}

```

Figura 14. Logs del operador de planificadores.

Lanzamos un planificador para ver que nuestro operador funciona como se espera, el cual, como podemos ver en la Figura 15 se ha desplegado correctamente.

```

Defaulted container "metricscheduler" out of: metricscheduler, k8tz (init)
/scheduler --metric-name=node_network_transmit_bytes_total --metric-start-date=now() - INTERVAL '2 DAYS' --metric-end-date=now
ority-order=desc --metric-is-second-level=false --metric-filter-clause=val(device_id)in('eth1') --metric-second-level-group='
db-host=timescaledb.monitoring --timescaledb-port=5432 --timescaledb-user=postgres --timescaledb-password=patroni --timescale
--scheduler-name=metricscheduler --log-level=info --filtered-nodes=tfm-dev --timeout=20
Scheduler started!
Config: {MetricParams:{MetricName:node_network_transmit_bytes_total StartDate:now() - INTERVAL '2 DAYS' EndDate:now() Operatio
val(device_id)in('eth1')} IsSecondLevel:false SecondLevelGroup: SecondLevelSelect:} TimescaledbParams:{Host:timescaledb.monito
Database:tsdb AuthenticationType:MD5} SchedulerName:metricscheduler Timeout:20 LogLevel:info FilteredNodes:tfm-dev}
{"level":"info","msg":"New Node Added to Store: tfm-dev","time":"2023-06-29T06:59:57Z"}
{"level":"info","msg":"New Node Added to Store: tfm-dev-m02","time":"2023-06-29T06:59:57Z"}
{"level":"info","msg":"New Node Added to Store: tfm-dev-m03","time":"2023-06-29T06:59:57Z"}
{"level":"info","msg":"New Node Added to Store: tfm-dev-m04","time":"2023-06-29T06:59:57Z"}
{"level":"info","msg":"New Node Added to Store: tfm-dev-m05","time":"2023-06-29T06:59:57Z"}
{"level":"info","msg":"New Node Added to Store: tfm-dev-m06","time":"2023-06-29T06:59:57Z"}

```

Figura 15. Log planificador.

Para finalizar esta prueba funcional lanzaremos un *pod* a través de este planificador (ver Código 10). Para ello indicaremos en el campo *Spec.SchedulerName* el nombre de nuestro planificador, *metricscheduler*.

```

apiVersion: v1
kind: Pod
metadata:
  name: pod01
  labels:
    name: pod01
spec:
  schedulerName: metricscheduler
  containers:
  - name: pod01
    image: registry.k8s.io/pause:2.0
    resources:
      requests:
        cpu: 500m
        memory: 512M
      limits:
        cpu: 500m
        memory: 512M

```

Código 10. Descriptor del *pod* de prueba

Como podemos ver en los *logs* el *pod* es planificado correctamente (Figura 16).

```

select row_number() over() rowid, a.node, a.value from (select left(val(instance_id),-5) node, max(value) value
' and time >=now()- INTERVAL '2 DAYS' and time <=now() and val(device_id)in('eth1') group by val(instance_id)
Values for metric node_network_transmit_bytes_total
Node 192.168.59.248 , metric value 5.57567055e+08
Node 192.168.59.246 , metric value 2.0311206e+08
Node 192.168.59.247 , metric value 6.684173e+06
Node 192.168.59.244 , metric value 4.815491e+06
Node 192.168.59.245 , metric value 4.772811e+06
Node 192.168.59.252 , metric value 4.566463e+06
{"level":"info","msg":"calculated priorities after filter nodes where pod fit: map[192.168.59.244:4 192.168.59.
,"time":"2023-06-29T07:01:25Z"}
{"level":"info","msg":"bestNode tfm-dev-m06 bestNodeIp: 192.168.59.252","time":"2023-06-29T07:01:25Z"}
Placed pod [ns1/pod01] on tfm-dev-m06

```

Figura 16. Logs del planificador tras lanzar el pod

Por último, vemos que en el *namespace* ns1 tenemos el operador, el planificador y el *pod* ejecutándose (ver Figura 17).

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
metricscheduler-859c86b4b4-9d2bz	1/1	Running	0	3m6s	10.244.3.3	tfm-dev-m04
pod01	1/1	Running	0	86s	10.244.5.3	tfm-dev-m06
scheduler-operator-694bfcfb4d-7k9xg	1/1	Running	0	3m21s	10.244.4.3	tfm-dev-m05

Figura 17. Pods del namespace ns1

Una vez validada la prueba funcional eliminaremos los componentes del *namespace* ns1: el *pod* creado, el planificador y el operador.

4.2 EVALUACIÓN COMPARATIVA DE PRESTACIONES

En esta segunda sección, vamos a comparar el funcionamiento de distintos algoritmos de planificación. Para ello, haremos uso de la flexibilidad y potencia de nuestro planificador para poder realizar cálculos de formas muy diferentes. Hemos diseñado los siguientes algoritmos para la realización de nuestras pruebas:

- Algoritmo *network aware*: en este caso, nuestro planificador priorizará nodos con baja transmisión de datos en su interfaz de red. Para ello, nos fijaremos en los cambios producidos en la métrica *node_network_transmit_bytes_total* en un determinado periodo de tiempo.
- Algoritmo *CPU aware*: el planificador estudiará la métrica *node_cpu_seconds_total*. Calculará en cuál de los nodos la suma de los tiempos en modo *idle* de cada una de sus *CPUs* es mayor, es decir que nodo ha tenido más tiempo sus *CPUs* en modo en espera.

Además, en esta evaluación se comparará con el algoritmo de planificación por defecto de Kubernetes y el planificador aleatorio que vimos anteriormente. Con ello, hemos definido los siguientes escenarios de evaluación sobre el entorno local descrito en el Capítulo 3:

- Escenario 1: Planificador por defecto de Kubernetes
- Escenario 2: Planificador aleatorio.
- Escenario 3: Planificador desarrollado priorizando nodos con baja transmisión de datos de red.
- Escenario 4: Planificador desarrollado priorizando nodos con alto tiempo de procesador en modo *idle* (en espera).

4.2.1 Entorno de evaluación y carga aplicada.

El entorno que vamos a utilizar para realizar la evaluación del planificador está formado por 5 nodos de Kubernetes con 4GB de *RAM* y 4 *CPUs* por cada uno de ellos. Debemos notar que el nodo *tfm-dev* incluye el *control plane* y el nodo *tfm-dev-m02* los componentes de monitorización.

Es por ello por lo que esos dos nodos *tfm-dev* y *tfm-dev-m02* no se usarán en las pruebas de carga y despliegue de *pods* con el planificador. Tras la instalación de todos los componentes de Minikube, los nodos que usaremos para realizar las evaluaciones han quedado con los recursos ocupados que se pueden ver en la Tabla 7.

NODO	CPU	MEMORIA
tfm-dev(control plane)	1	220Mi
tfm-dev-m02 (monitoring)	250m	50Mi
tfm-dev-m03	250m	50Mi
tfm-dev-m04	250m	50Mi
tfm-dev-m05	250m	50Mi

Tabla 7. Recursos ocupados por nodo tras la instalación.

Para realizar la evaluación de los planificadores lanzaremos baterías de *pods* con las mismas características para poder estudiar cómo son desplegados de distinta forma dependiendo del algoritmo utilizado. El *pod* que se lanzará se puede ver en el Código 10.

4.2.2 Test de carga del clúster

Previo al lanzamiento de las baterías de *pods* se estresará el clúster para disponer de un entorno donde se haya realizado una carga real tanto del tráfico de red como del consumo de *CPUs* de los distintos nodos.

Para este punto, se ha buscado algún tipo de herramienta que genere carga en el clúster de forma diferente a la que se obtendría levantando una aplicación, como por ejemplo tráfico de red entre los nodos. Como vimos en el análisis, un conjunto de métricas que nos interesaban eran las relacionadas con el consumo de red. En este punto y tras barajar diversas opciones, se decidió desplegar un pequeño clúster de Kafka.

Apache Kafka [38] es una plataforma distribuida para la transmisión de datos de datos que permite no solo publicar, almacenar y procesar flujos de eventos de forma inmediata, sino también para suscribirse a ellos. Está diseñada para administrar los flujos de varias fuentes y enviarlos a distintos usuarios. Kafka hace uso de Zookeeper [39] para coordinar sus procesos y el estado del clúster. Zookeeper es un servicio para la coordinación de procesos distribuido y altamente confiable para sistemas distribuidos

En este proyecto se ha usado para generar escenarios de carga para que el planificador disponga de datos sobre los cuales realizar sus funciones. Además, se creó un pequeño contenedor con una herramienta propia de Apache que sirve para lanzar pruebas de rendimiento y que se basa en un productor o un consumidor.

Entonces, teniendo este pequeño clúster de Kafka levantado y una aplicación, produciendo, podremos aumentar la transmisión de datos de red. Como premisa suponemos que el nodo donde se despliegue este productor debería ser el que más datos transmitiera ya que en el periodo de estudio el productor estará continuamente mandando mensajes a Kafka.

Por ello, sabiendo en qué nodos están estas aplicaciones, podremos comparar con los datos obtenidos de Prometheus y persistidos en TimescaleDB para validar que arquitectura diseñada funciona como se espera, para posteriormente realizar pruebas sabiendo que los datos son confiables.

Por lo tanto, tenemos un clúster de Kafka desplegado con el operador de Strimzi [40] con el productor enviando mensajes al clúster de Kafka (ver Figura 18).

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
kafka-cluster-entity-operator-6dcd97b44b-7fm56	3/3	Running	0	2m22s	10.244.4.4	tfm-dev-m05
kafka-cluster-kafka-0	1/1	Running	0	3m17s	10.244.3.4	tfm-dev-m04
kafka-cluster-zookeeper-0	1/1	Running	0	4m11s	10.244.2.4	tfm-dev-m03
kafka-producer-8d4dd8cb7-z8tgn	1/1	Running	0	6s	10.244.4.5	tfm-dev-m05
strimzi-cluster-operator-77f8975f54-zx75x	1/1	Running	0	5m20s	10.244.4.3	tfm-dev-m05

Figura 18. Despliegue del clúster de *Kafka*

Tras un periodo de carga de una hora que garantiza la estabilidad del sistema y la persistencia de las métricas necesarias para los planificadores diseñados, comprobaremos su funcionamiento consultando la transmisión de bytes de cada uno de los nodos. Filtraremos los nodos de Kubernetes y monitorización para realizar nuestras pruebas.

```
SELECT left(Val(instance_id),-5) node,
       Last(value,time) - first(value,time) value
FROM   node_network_transmit_bytes_total
WHERE  value <> 'Nan'
AND    time >=Now() - interval '15 MINUTES'
AND    time <=now()
AND    val(device_id) IN ('eth1')
AND    left(Val(instance_id),-5) <> '192.168.59.253'
AND    left(Val(instance_id),-5) <> '192.168.59.254'
GROUP BY val(instance_id)
ORDER BY Last(value,time) - first(value,time) DESC
```

Código 11. Consulta de transmisión de datos en cada nodo.

node	value
192.168.59.250	552534051
192.168.59.251	64911395
192.168.59.252	11261187

Figura 19. Resultados de la consulta de transmisión de datos en cada nodo.

Tras ejecutar la consulta del Código 11. Consulta de transmisión de datos en cada nodo., podemos ver en la Figura 19. que es el nodo con IP 192.168.59.250 el que más bytes ha transmitido con una diferencia muy significativa sobre los demás. Si comprobamos en qué nodo está esa IP en la captura del *namespace* de monitorización, vemos que es el *tfm-dev-m5*. Si a su vez comprobamos en qué nodo está el *pod* productor en la captura del *namespace* de *Kafka*, vemos que es el mismo.

Por lo tanto, vemos que se cumple la premisa que habíamos supuesto de que el nodo donde se encontrase el productor debería ser el que más datos de red transmitiera, ya que ha estado continuamente enviando mensajes al clúster.

También vamos a estudiar el tiempo que han estado las *CPUs* de cada nodo en modo *idle*. En las Figura 20, Figura 21 y Figura 22 podemos ver los recursos ocupados en los nodos *tfm-dev-m03*, *tfm-dev-m04* y *tfm-dev-m05* respectivamente.

Resource	Requests
-----	-----
cpu	1250m (31%)
memory	1074Mi (27%)
ephemeral-storage	0 (0%)
hugepages-2Mi	0 (0%)

Figura 20. Ocupación tfm-dev-m03

Resource	Requests
-----	-----
cpu	1250m (31%)
memory	1074Mi (27%)
ephemeral-storage	0 (0%)
hugepages-2Mi	0 (0%)

Figura 21. Ocupación tfm-dev-m04

Resource	Requests
-----	-----
cpu	1950m (48%)
memory	1479081984 (35%)
ephemeral-storage	0 (0%)
hugepages-2Mi	0 (0%)

Figura 22. Ocupación tfm-dev-m05

Además, en la Figura 18, veíamos qué componentes se estaban ejecutando en cada uno de los nodos:

- *tfm-dev-m03* (192.168.59.252): Clúster de Zookeeper
- *tfm-dev-m04* (192.168.59.251): Clúster de Kafka.
- *tfm-dev-m05* (192.168.59.250): Productor, operador y otro proceso del operador llamado *kafka-cluster-entity-operator*.

Con estos datos cabría pensar que el nodo que estará más ocupado y por tanto sus *CPUs* estarán menos tiempo en modo *idle* debería ser *tfm-dev-m05*, ya que tiene más recursos asignados y más procesos corriendo. En segundo lugar, debería ser *tfm-dev-m04* ya que, aunque tienen los mismos recursos asignados que *tfm-dev-m03*, tiene corriendo un clúster de Kafka que debería estar más cargado a nivel de *CPU* que el clúster de Zookeeper, ya que está recibiendo constantemente información y aunque Zookeeper también estará recibiendo metadatos serán menos pesados y más fáciles de procesar que los mensajes. Por ello el nodo con más tiempo en *idle* debería ser *tfm-dev-m03*.

Tras ejecutar la consulta del Código 12. Consulta de *CPUs* en estado *idle* en cada nodo., podemos ver en la Figura 23 que nuestras premisas se han cumplido. Con todo ello, podemos dar por validada la solución. Antes de lanzar las pruebas, se eliminarán los componentes de Kafka para maximizar el espacio disponible en los nodos.

```

SELECT  left(a.node,-5) node,
Sum(a.value)          value
FROM    (
SELECT  Val(instance_id)      node,
val(cpu_id) cpu,
Last(value,time) - first(value,time) value
FROM    node_cpu_seconds_total
WHERE    value <> 'Nan'
AND      time >=Now() - interval '15 MINUTE'
AND      time <=now() - interval '5 MINUTE'
AND      val(mode_id)='idle'
AND      left(Val(instance_id),-5) <> '192.168.59.253'
AND      left(Val(instance_id),-5) <> '192.168.59.254'
GROUP BY val(instance_id),
val(cpu_id)) a
GROUP BY node
ORDER BY sum(value) ASC

```

Código 12. Consulta de CPUs en estado *idle* en cada nodo.

node	value
192.168.59.250	2744.4599999999996
192.168.59.251	2805.35
192.168.59.252	3087.1000000000004

Figura 23. Resultados de la consulta de CPUs en estado *idle* en cada nodo

4.2.3 Análisis de resultados

Vamos a lanzar una batería de 20 *pods* con las características que se indicaron en la prueba funcional (media CPU y 512 MB RAM), para ver cómo se reparten entre los nodos en cada uno de los escenarios. Como requisito previo, en cada uno de los planificadores se ha restringido el lanzamiento de los *pods*, no permitiéndose que se ejecuten en los dos nodos donde se encuentran los componentes de Kubernetes y los de monitorización.

Para el caso del planificador por defecto, en cada uno de los *pods* lanzados se ha añadido una restricción *Affinity* con el operador *NotIn* para los nodos *tfm-dev* y *tfm-dev-m02*. En el resto de los escenarios se modificó el código del planificador para poder filtrar los nodos no deseados.

4.2.3.1 Escenario 1: Planificador por defecto

En este primer escenario se ha utilizado el planificador por defecto de Kubernetes. Como podemos ver en la Figura 24 el planificador por defecto reparte los *pods* de forma totalmente equitativa entre cada uno de los nodos.

Además, en la Figura 25, vemos el orden en el que se han lanzado los *pods*. En este caso al tener todos los nodos la misma capacidad inicial, el planificador ha ido repartiéndolos de modo equitativo entre todos los nodos con la finalidad de ir ocupándolos de un modo lo más uniforme posible siguiendo para ello un modelo de asignación *round-robin*.

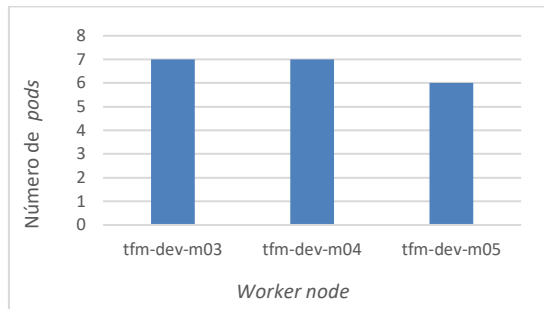


Figura 24. E1: número de *pods* en cada nodo

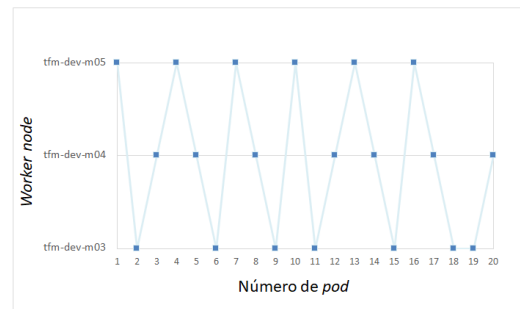


Figura 25. E1: Ejecución temporal de los *pods*

Por otro lado, en las Figura 26 y Figura 27, vemos que tanto el porcentaje de uso de *CPU* como la memoria usada en cada *worker node* se han repartido de la misma manera. Estas gráficas siguen el mismo perfil que la de la asignación de *pods* por nodo debido a que todos los *pods* tienen las mismas características. Dado que estas dos gráficas no muestran información significativa no se mostrarán en el resto de los escenarios.

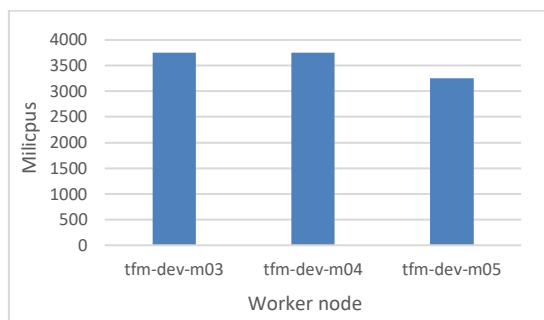


Figura 26. E1: CPU consumida en cada nodo (mili)

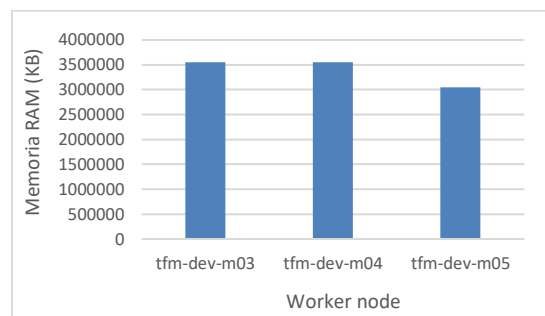


Figura 27. E1: Memoria consumida en cada nodo (KB)

4.2.3.2 Escenario 2: Planificador aleatorio

En este segundo escenario vamos a hacer uso del planificador aleatorio del que hablamos en secciones anteriores.

En la Figura 29 se puede ver que en este caso no se ha seguido ningún orden para la asignación de los *Pods*, si no que ésta se ha realizado de forma aleatoria. Además, como este planificador no filtraba por recursos disponibles en el nodo, vemos que en uno de ellos se han asignado más *Pods* que la capacidad disponible en el nodo (*tfm-dev-m04*) y por lo tanto se ha producido un error *OutOfCPU*, el cual podemos ver en la Figura 28, ya que el nodo no tenía más capacidad disponible.

pod15	1/1	Running	0	6m7s	10.244.3.74	tfm-dev-m04
pod16	0/1	OutOfcpu	0	3m26s	<none>	tfm-dev-m04
pod17	1/1	Running	0	6m6s	10.244.3.75	tfm-dev-m04
pod18	1/1	Running	0	6m5s	10.244.2.78	tfm-dev-m03
pod19	1/1	Running	0	6m4s	10.244.4.77	tfm-dev-m05
pod20	0/1	OutOfcpu	0	6m4s	<none>	tfm-dev-m04

Figura 28. *Pods* no asignados por error *OutOfcpu*

En la Figura 30 se puede ver el orden en el que se han lanzado los *Pods* y como el algoritmo sigue una lógica aleatoria. Los *Pods* que dieron error *OutOfCPU* no se intentaron reasignar debido a que el planificador aleatorio no tenía capacidad para ello, por lo que éste se quedó en un estado erróneo.

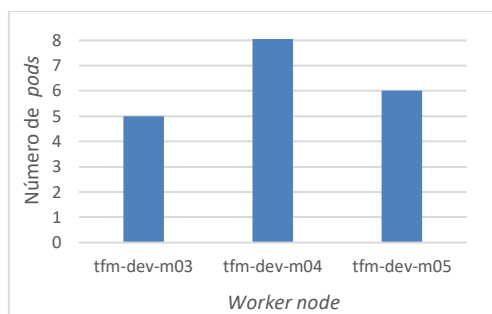


Figura 29. E2: número de *Pods* en cada nodo

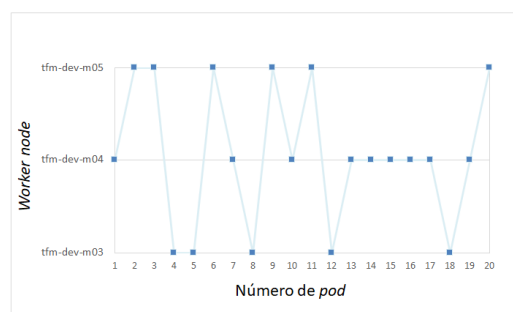


Figura 30. E2: Ejecución temporal de los *Pods*

4.2.3.3 Escenario 3: Planificador priorizando la asignación en nodos con bajo tráfico saliente de red.

Este escenario se lanzará en el clúster donde se había lanzado la prueba de carga y tras eliminar los componentes de Kafka. El planificador se desplegará una vez levantado el operador con el *script* que se puede ver en el Anexo A.7.

En la Figura 31 podemos ver cómo han quedado los *pods* repartidos en los nodos al finalizar el lanzamiento. Además, en la Figura 32 se muestra la ejecución a lo largo del tiempo de los *pods* y como se han ido asignando al nodo correspondiente hasta que se ha llenado para continuar después con el siguiente nodo asignable. El orden de asignación concuerda con el que vimos en la Figura 19 y también con cómo lo previmos tras el lanzamiento de la prueba de carga.

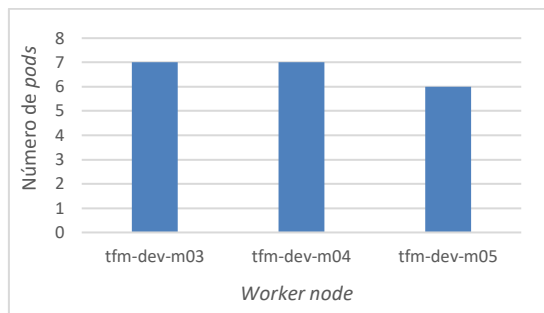


Figura 31. E3: número de *pods* en cada nodo

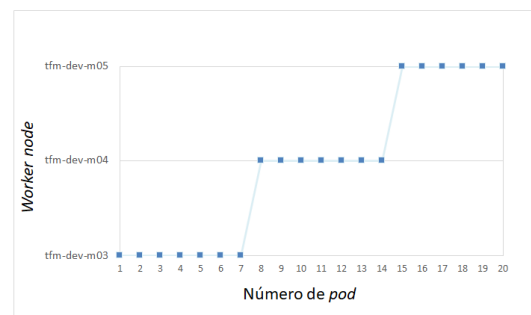


Figura 32. E3: ejecución temporal de los *pods*

4.2.3.4 Escenario 4: Planificador priorizando la asignación en nodos con mucho tiempo de CPU en espera.

Este escenario se lanzará en el clúster donde se había lanzado la prueba de carga y tras eliminar los componentes de Kafka. El planificador se lanzará una vez levantado el operador con el *script* que se puede ver en el Anexo A.9.

En la Figura 33 podemos ver cómo han quedado los *pods* repartidos en los nodos al finalizar el lanzamiento. Además, en la Figura 34 se muestra la ejecución a lo largo del tiempo de los *pods* y como se han ido asignando al nodo correspondiente hasta que se ha

llenado para continuar después con el siguiente nodo asignable. El orden de asignación concuerda con el que vimos en la Figura 23, es decir llenando nodos con mayor tiempo de *CPU* en modo *idle*.

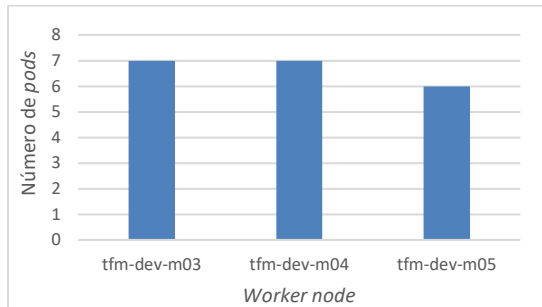


Figura 33. E3: número de *pods* en cada nodo

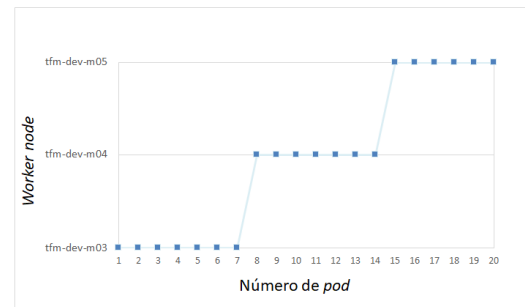


Figura 34. E3: ejecución temporal de los *pods*

CAPÍTULO 5. CONCLUSIONES Y TRABAJO FUTURO

En este último capítulo en el que concluimos el desarrollo de este Trabajo Fin de Máster, se van a detallar las conclusiones más importantes a las que se han llegado tras la finalización de este. Además, se enumerarán posibles mejoras y ampliaciones futuras que se podrían realizar sobre el mismo.

5.1 CONCLUSIONES

La realización de este Trabajo Fin de Máster me pareció muy interesante desde el primer momento ya que todo lo relacionado con entornos distribuidos me atrae mucho. Sobre todo, comprender como interaccionan las distintas partes de estos sistemas, poder expandirlas y desarrollar sobre ellas.

Desde hace años me dedico profesionalmente al desarrollo dentro de este tipo de sistemas con un perfil de DevOps, actualmente trabajando con Kubernetes, pero años atrás también con Apache Mesos. Esto me ha facilitado la comprensión de las etapas iniciales del proyecto y el poder resolver de una manera más eficaz problemas que en caso de haber empezado de cero me habría llevado mucho más trabajo.

Aun así, al ser Kubernetes un sistema tan grande, la realización de este trabajo me ha permitido profundizar en aspectos que no conocía, como por ejemplo el desarrollo de

planificadores personalizados. También me ha parecido realmente interesante el poder crear un sistema con tantas piezas interactuando entre sí: Kubernetes, el servidor de métricas, TimescaleDB, Promscale, Kafka, nuestro propio operador desarrollado con Kubebuilder y el planificador interactuando con ellas.

Centrándonos en los objetivos especificados en el Capítulo 1, éstos han sido alcanzados de manera satisfactoria, dando lugar a un buen punto de partida para el planificador que podría ser ampliado en un futuro.

También habría que comentar que a lo largo del trabajo surgieron diversos problemas que se detallaron en el Apartado 3. Lo cual se ha visto reflejada en sobrepasar levemente la estimación inicial que veíamos en la Figura 1. Éstos deben ayudarnos a entender que la realización de cualquier proyecto no está exenta de dificultades inesperadas y que debemos estar preparados para poder resolverlas de la manera más efectiva.

Con todo esto, me gustaría concluir diciendo que el desarrollo de este Trabajo Fin de Máster ha sido una grata experiencia que me permitirá enfrentarme a nuevos retos en el futuro.

5.2 TRABAJO FUTURO Y POSIBLES AMPLIACIONES

Para finalizar con este último capítulo, comentaremos algunas de las propuestas que se podrían añadir al trabajo actual, en trabajos futuros:

- Propuesta 1. Ampliar el catálogo de algoritmos que se pueden lanzar con el planificador. Como se ha visto en los apartados anteriores el planificador es capaz de realizar cálculos de hasta dos niveles de agrupación de datos. Sin embargo, esta capacidad es fácilmente ampliable y personalizable. Por ello una propuesta muy interesante sería crear nuevos algoritmos que pudiesen realizar cálculos más complejos. También se le podría incluir algún motor de inteligencia artificial para que generase modelos predictivos con los datos persistidos en TimescaleDB.

- Propuesta 2. El estado actual del trabajo es lo que empresarialmente se conoce como *Project Ready* debido a que es un Trabajo Fin de Máster. Es por ello por lo que otra mejora interesante sería el ampliar funcionalidades al operador que para nuestro propósito no han sido necesarias, como por ejemplo que se pudiese comprobar en todo momento el estado del *Custom Resource*, añadir restricciones y valores por defecto al CR mediante los *webhooks* tanto en su creación como en su actualización, ... En definitiva, llevar el proyecto a un nivel *Production Ready*.

BIBLIOGRAFÍA

LIBROS Y ARTICULOS

- [Car22] Carmen Carrión. 2022. Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges. ACM Comput. Surv. 55, 7, Article 138 (July 2023), 37 pages. <https://doi.org/10.1145/3539606>
- [Zei22] Zeineb Rejiba and Javad Chamanara. 2022. Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches. ACM Comput. Surv. 55, 7, Article 151 (July 2023), 37 pages. <https://doi.org/10.1145/3544788>
- [Tur14] Turnbull, J., 2014. The Docker Book: Containerization is the new virtualization. James Turnbull. <https://www.oreilly.com/library/view/the-docker-book/9780988820203/>
- [Nic19] Nickoloff, J. and Kuenzli, S., 2019. Docker in action. Simon and Schuster. <https://www.simonandschuster.com/books/Docker-in-Action-Second-Edition/Jeffrey-Nickoloff/9781638351740>
- [Say17] Sayfan, G., 2017. Mastering kubernetes. Packt Publishing Ltd. <https://www.packtpub.com/product/mastering-kubernetes/9781786461001>
- [Dob20] Dobies, J., & Wood, J. (2020). Kubernetes Operators: Automating the Container Orchestration Platform. " O'Reilly Media, Inc.". Disponible en: <https://www.oreilly.com/library/view/kubernetes-operators/9781492048039/>

ENLACES INTERNET

- [1] Cloud Fundation. Kubernetes. Último acceso: junio 9, 2023
<https://www.cncf.io/projects/kubernetes/>
- [2] Kubernetes – Pod. Último acceso: junio 22, 2023
<https://kubernetes.io/docs/concepts/workloads/pods/>
- [3] Kubernetes - Architecture Overview. Último acceso: junio 9, 2023
<https://medium.com/devops-mojokubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd>
- [4] Kubernetes - Controller. Último acceso: junio 22, 2023
<https://kubernetes.io/docs/concepts/architecture/controller/>
- [5] Kubernetes – ReplicaSets. Último acceso: junio 22, 2023
<https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/>
- [6] Kubernetes – Deployments. Último acceso: junio 22, 2023
<https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
- [7] Operator SDK. Último acceso: junio 8, 2023
<https://sdk.operatorframework.io/>
- [8] Kubebuilder. Github. Último acceso: junio 8, 2023
<https://github.com/kubernetes-sigs/kubebuilder>
- [9] Everything You Need To Know About Kubernetes Scheduler. Último acceso: junio 9, 2023
<https://totalcloudio.medium.com/everything-you-need-to-know-about-kubernetes-scheduler-b1e67a4257f1>
- [10] Kubernetes Scheduler. Último acceso: junio 9, 2023
<https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>
- [11] The Kubernetes Scheduler. Último acceso: junio 9, 2023
<https://dominik-tornow.medium.com/the-kubernetes-scheduler-cd429abac02f>
- [12] Configure Multiple Schedulers. Último acceso: junio 9, 2023
<https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/>

- [13] Random Kubernetes scheduler. Último acceso: junio 9, 2023
<https://github.com/martonsereg/random-scheduler>
- [14] A Deep Dive Into Kubernetes Informers. Último acceso: julio 2, 2023
<https://aly.arriqaaq.com/kubernetes-informers/>
- [15] Kind. Último acceso: junio 9, 2023
<https://kind.sigs.k8s.io/>
- [16] Minikube. Documentation. Último acceso: junio 9, 2023
<https://minikube.sigs.k8s.io/docs/start/>
- [17] Prometheus. Overview. Último acceso: junio 8, 2023
<https://prometheus.io/docs/introduction/overview/>
- [18] Monitoring Linux host metrics with the node exporter. Último acceso: junio 10, 2023
<https://prometheus.io/docs/guides/node-exporter/>
- [19] How to Setup Prometheus Monitoring On Kubernetes Cluster. Último acceso: junio 10, 2023
<https://devopscube.com/setup-prometheus-monitoring-on-kubernetes/>
- [20] How to Setup Prometheus Node Exporter on Kubernetes. Último acceso: junio 10, 2023
<https://devopscube.com/node-exporter-kubernetes/>
- [21] Network interface metrics from the node exporter. Último acceso: junio 10, 2023
<https://www.robustperception.io/network-interface-metrics-from-the-node-exporter/>
- [22] Mapping iostat to the node exporter's node_disk_* metrics. Último acceso: junio 10, 2023
https://www.robustperception.io/mapping-iostat-to-the-node-exporters-node_disk_-metrics/
- [23] Understanding Machine CPU usage. Último acceso: junio 10, 2023
<https://www.robustperception.io/understanding-machine-cpu-usage>
- [24] How to Deploy PostgreSQL on Kubernetes. Último acceso: junio 11, 2023
<https://phoenixnap.com/kb/postgresql-kubernetes>

- [25] PostgreSQL Prometheus Adapter. Último acceso: junio 11, 2023
<https://github.com/CrunchyData/postgresql-prometheus-adapter>
- [26] Timescale: Time-series data simplified. Último acceso: junio 11, 2023
<https://www.timescale.com>
- [27] Promscale documentation. Último acceso: junio 11, 2023
<https://promscale-legacy-docs.timescale.com/>
- [28] Promscale: An analytical platform and long-term store for Prometheus, with the combined power of SQL and PromQL. Último acceso: junio 22, 2023
<https://www.timescale.com/blog/promscale-analytical-platform-long-term-store-for-prometheus-combined-sql-promql-postgresql/>
- [29] Helm. El administrador de paquetes para Kubernetes. Último acceso: junio 24, 2023
<https://helm.sh/es/>
- [30] Perform advanced analytic queries. Último acceso: junio 12, 2023
<https://docs.timescale.com/use-timescale/latest/query-data/advanced-analytic-queries/>
- [31] Custom Kubernetes Scheduler. Último acceso: junio 12, 2023
<https://docs.timescale.com/use-timescale/latest/query-data/advanced-analytic-queries/>
- [32] Random Kubernetes scheduler. Último acceso: junio 12, 2023
<https://github.com/martonsereg/random-scheduler>
- [33] Documentation. The Go Programming Language. Último acceso: junio 8, 2023
<https://go.dev/doc/>
- [34] Kubebuilder. Github. Último acceso: junio 12, 2023
<https://github.com/kubernetes-sigs/kubebuilder>
- [35] Extend the Kubernetes API with CustomResourceDefinitions. Último acceso: junio 12, 2023
<https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/>
- [36] Kubernetes. Controller-tools library. Último acceso: junio 24, 2023
<https://github.com/kubernetes-sigs/controller-tools>

- [37] Kubernetes. Custom Resources. Último acceso: junio 24, 2023
<https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>
- [38] Apache Kafka. Último acceso: junio 27, 2023
<https://kafka.apache.org/>
- [39] Welcome to Apache Zookeeper. Último acceso: julio 5, 2023
<https://zookeeper.apache.org/>
- [40] Strimzi. Último acceso: junio 27, 2023
<https://strimzi.io/>

GLOSARIO DE TÉRMINOS

API Server

Es un componente del *control plane* que expone la API de Kubernetes. Se puede considerar como el *front-end* del *control plane*.

Control plane

Es la capa de orquestación de contenedores que expone la API y las interfaces para definir, implementar y administrar el ciclo de vida de los contenedores.

Kubelet

Es un agente que se ejecuta en cada nodo del clúster. Se asegura que los contenedores se ejecuten en un *pod*. No gestiona contenedores que no hayan sido creados mediante Kubernetes.

Namespace

Kubernetes soporta múltiples clústeres virtuales respaldados por el mismo clúster físico. Estos clústeres virtuales se denominan espacios de nombres (*namespaces*).

ANEXO A. *SCRIPTS* DE INSTALACIÓN DE LOS COMPONENTES DEL CLÚSTER

A.1 INTRODUCCIÓN

En este anexo se adjuntam los *scripts* de instalación de diversos componentes del proyecto. Para ver el código en mayor profundidad se puede consultar el repositorio de código que se adjunta a este trabajo, el cual se detalla en el Anexo B.

A.2 INSTALACIÓN DEL CLÚSTER MEDIANTE KIND

Lo primero que hicimos fue crear un registro local donde almacenar las imágenes de Docker, como se puede ver en Código 13.

```
reg_name='kind-registry'
reg_port='5000'
running="$(docker inspect -f '{{.State.Running}}' "${reg_name}" 2>/dev/null || true)"

if [ "${running}" != 'true' ]; then
  docker run \
    -d --restart=always -p "${reg_port}:5000" --name "${reg_name}" \
    registry:2
fi
```

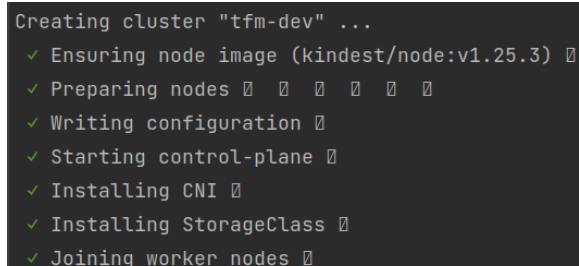
Código 13. Creación de registro local de Docker

Posteriormente se lanzó el clúster mediante el *script* que podemos ver en Código 14.

```
cat <<EOF | kind create cluster --name="${cluster_name}" --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
containerdConfigPatches:
- |-
[plugins."io.containerd.grpc.v1.cri".registry.mirrors."localhost:${reg_port}"]
  endpoint = ["http://${reg_name}:${reg_port}"]
nodes:
- role: control-plane
  image: ${kindDocker}
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
- role: worker
  image: ${kindDocker}
- role: worker
  image: ${kindDocker}
- role: worker
  image: ${kindDocker}
- role: worker
  image: ${kindDocker}
- role: worker
  image: ${kindDocker}
```

Código 14. *Script* de creación del clúster mediante Kind

Obteniendo a partir de ello un clúster de Kubernetes local y funcional como vemos en Figura 35.



```
Creating cluster "tfm-dev" ...
✓ Ensuring node image (kindest/node:v1.25.3) 
✓ Preparing nodes 
✓ Writing configuration 
✓ Starting control-plane 
✓ Installing CNI 
✓ Installing StorageClass 
✓ Joining worker nodes
```

Figura 35. Clúster de Kubernetes creado con Kind

A.3 INSTALACIÓN DEL CLÚSTER MEDIANTE MINIKUBE

Podemos ver el *script* de creación del entorno en Código 15.

```
echo "Deploying Kubernetes"
cluster_name='tfm-dev'

minikube start --cpus 4 --memory 4072 --nodes 5 -p ${cluster_name} --driver=virtualbox --
insecure-registry 192.168.59.100:5000 --disk-size 50GB

minikube addons enable registry -p ${cluster_name}
```

Código 15. *Script* de creación del entorno con Minikube

El cual nos crea un entorno con 5 nodos virtuales, con 4 *CPUs*, 4 GB de *RAM* y 50 GB de disco para cada uno de ellos como vemos en Figura 36.

NAME	STATUS	ROLES	AGE	VERSION
tfm-dev	Ready	control-plane	20m	v1.26.3
tfm-dev-m02	Ready	<none>	19m	v1.26.3
tfm-dev-m03	Ready	<none>	18m	v1.26.3
tfm-dev-m04	Ready	<none>	17m	v1.26.3
tfm-dev-m05	Ready	<none>	16m	v1.26.3

Figura 36. Clúster de Kubernetes creado con Minikube

A.4 INSTALACIÓN DEL SERVIDOR DE MÉTRICAS

Una vez desplegado el servidor comprobamos que se ha desplegado correctamente con Código 16 en la Figura 37.

```
kubectl get pods -n monitoring
```

Código 16. Obtención de *pods* del *namespace* de monitorización

NAME	READY	STATUS	RESTARTS	AGE
node-exporter-9wnnk	1/1	Running	0	9m32s
node-exporter-ft99c	1/1	Running	0	9m32s
node-exporter-hv64j	1/1	Running	0	9m32s
node-exporter-lq22s	1/1	Running	0	9m32s
node-exporter-xvcts	1/1	Running	0	9m32s
prometheus-deployment-67cf879cc4-wcdl5	1/1	Running	0	9m32s

Figura 37. Servidor de monitorización desplegado

Y podemos ver en la Figura 38. Resultado de consulta de métricas como el servidor devuelve las métricas que necesitamos ejecutando el Código 17. *Script* para obtener métricas.

```
kubectl port-forward service/node-exporter -n monitoring 9100:9100
curl http://localhost:9100/metrics
```

Código 17. *Script* para obtener métricas

```
# HELP node_network_speed_bytes Network device property: speed_bytes
# TYPE node_network_speed_bytes gauge
node_network_speed_bytes{device="docker0"} -125000
node_network_speed_bytes{device="eth0"} -125000
node_network_speed_bytes{device="eth1"} -125000
# HELP node_network_transmit_bytes_total Network device statistic transmit_bytes.
# TYPE node_network_transmit_bytes_total counter
node_network_transmit_bytes_total{device="docker0"} 0
node_network_transmit_bytes_total{device="eth0"} 2.305999e+06
node_network_transmit_bytes_total{device="eth1"} 9.355598e+06
node_network_transmit_bytes_total{device="lo"} 5146
node_network_transmit_bytes_total{device="sit0"} 0
node_network_transmit_bytes_total{device="veth432778d9"} 357676
node_network_transmit_bytes_total{device="veth57d4aea4"} 3177
node_network_transmit_bytes_total{device="veth9275c55d"} 132914
# HELP node_network_transmit_carrier_total Network device statistic transmit_carrier.
# TYPE node_network_transmit_carrier_total counter
node_network_transmit_carrier_total{device="docker0"} 0
node_network_transmit_carrier_total{device="eth0"} 0
```

Figura 38. Resultado de consulta de métricas

A.5 ADAPTACIÓN DEL ADAPTADOR DE CRUNCHY ROLL PARA LANZARLO EN EL ENTORNO

Para adaptar la herramienta se llevaron a cabo los siguientes pasos:

- Se introdujo la herramienta dentro de un contenedor de Docker y se expusieron todos los parámetros de configuración para poder lanzarlo como un objeto *Deployment* de Kubernetes. El descriptor se puede ver en Código 18.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: monitoring
  name: prometheus-postgres-adapter
  labels:
    app: postgresql-prometheus-adapter
```

```
spec:
  selector:
    matchLabels:
      app: postgresql-prometheus-adapter
  replicas: 1
  template:
    metadata:
      labels:
        app: postgresql-prometheus-adapter
    spec:
      hostname: adapter
      subdomain: adapter-example
      containers:
        - name: prometheus-postgres-adapter
          image: localhost:5000/crunchydata/postgresql-prometheus-adapter:1.1
          imagePullPolicy: Always
          ports:
            - containerPort: 9201
              name: listen
              protocol: TCP
          env:
            - name: DATABASE_URL
              value: "postgres://prometheus:prometheus@postgres:5432/prometheus"
            - name: ADAPTER_SEND_TIMEOUT
              value: "30s"
            - name: WEB_LISTEN_ADDRESS
              value: ":9201"
            - name: WEB_TELEMETRY_PATH
              value: "/metrics"
            - name: LOG_LEVEL
              value: "info"
            - name: PG_PARTITIONS
              value: "hourly"
            - name: PG_COMMIT_SECS
              value: "60"
            - name: PG_COMMIT_ROWS
              value: "5000"
          resources:
            requests:
              cpu: 1500m
              memory: 3072M
            limits:
              cpu: 1500m
              memory: 3072M
```

Código 18. Descriptor del *Deployment* del *adapter*

- Se creó un objeto *Service* para que el adaptador tuviera acceso a Prometheus y viceversa, el descriptor se puede ver en Código 19.

```
apiVersion: v1
kind: Service
metadata:
  name: postgresql-prometheus-adapter
  namespace: monitoring
spec:
  selector:
    app: postgresql-prometheus-adapter
  ports:
    - port: 9201
      targetPort: 9201
```

Código 19. Descriptor del *Service* del *adapter*

A.6 SCRIPT DE INSTALACIÓN DE TIMESCALEDB Y PROMSCALE

La instalación de estos componentes se realizó mediante Helm. Como se puede ver en el *script* en Código 20 es necesario crear la base de datos *tsdb* antes de instalar Promscale.

```
helm repo add timescale 'https://charts.timescale.com'
helm repo update

echo "4.1 Installing Timescale server"

helm install timescaledb timescale/timescaledb-single -f
resources/dependencies/timescale/values.yaml -n monitoring
kubectl -n monitoring wait --for=condition=ready --timeout=120s pod -l app=timescaledb

kubectl -n monitoring exec -i --tty "$(kubectl get pod -o name --namespace monitoring -l
role=master,release=timescaledb)" -- psql -U postgres -c "CREATE DATABASE tsdb WITH OWNER
postgres;"

echo "4.2 Installing promscale"

helm install -n monitoring promscale timescale/promscale -f
resources/dependencies/promscale/values.yaml

kubectl -n monitoring wait --for=condition=ready --timeout=120s pod -l app=promscale
```

Código 20. *Script* de instalación de TimescaleDB y Promscale

A.7 SCRIPTS DE PLANIFICADORES EJECUTADOS EN LA SECCIÓN DE RESULTADOS.

Para el planificador del escenario 3 se usó el descriptor que se muestra en Código 21.

```
apiVersion: scheduler.uclm.es/v1
kind: MetricScheduler
metadata:
  name: metricscheduler
  namespace: ns1
spec:
  image: localhost:5000/albertogomez/scheduler:0.0.0
  imagePullPolicy: Always
  instances: 1
  resources:
    requests:
      cpu: "1"
      memory: "1024Mi"
    limits:
      cpu: "1"
      memory: "1024Mi"
  healthchecks:
    livenessProbe:
```

```

    initialDelaySeconds: 10
    periodSeconds: 10
  readinessProbe:
    initialDelaySeconds: 5
    periodSeconds: 10
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 45%
      maxUnavailable: 35%
  timescaledb:
    host: timescaledb.monitoring
    port: "5432"
    user: postgres
    password: patroni
    database: tsdb
    authenticationType: MD5
  metric:
    name: "node_network_transmit_bytes_total"
    startDate: "now()- INTERVAL '15 MINUTES'"
    endDate: "now()"
    operation: "Last(value;time)-first(value;time)"
    priorityOrder: "desc"
    deviceList: "eth1"
    isSecondLevel: false

```

Código 21. Descriptor del planificador para el escenario 3

Para el escenario 4 se ha usado el se ve en Código 22.

```

apiVersion: scheduler.uclm.es/v1
kind: MetricScheduler
metadata:
  name: metricscheduler
  namespace: ns1
spec:
  image: localhost:5000/albertogomez/scheduler:0.0.0
  imagePullPolicy: Always
  instances: 1
  resources:
    requests:
      cpu: "1"
      memory: "1024Mi"
    limits:
      cpu: "1"
      memory: "1024Mi"
  healthchecks:
    livenessProbe:
      initialDelaySeconds: 10
      periodSeconds: 10
    readinessProbe:
      initialDelaySeconds: 5
      periodSeconds: 10
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 45%
      maxUnavailable: 35%
  filterNodes:
    - tfm-dev
    - tfm-dev-m02
  timescaledb:
    host: timescaledb.monitoring
    port: "5432"
    user: postgres

```

```
password: patroni
database: tsdb
authenticationType: MD5
metric:
  name: "node_cpu_seconds_total"
  startDate: "now() - INTERVAL '45 MINUTE'"
  endDate: "now() - INTERVAL '5 MINUTE'"
  operation: "sum"
  priorityOrder: "asc"
  isSecondLevel: true
  secondLevelSelect:
    - "val(instance_id)node"
    - "val(cpu_id)cpu"
    - "Last(value;time)-first(value;time) value"
  secondLevelGroup:
    - "val(instance_id) "
    - "val(cpu_id) "
  filters:
    - "val(mode_id)='idle'"
```

Código 22. Descriptor del planificador para el escenario 4

ANEXO B. MATERIAL COMPLEMENTARIO

En este anexo se indicará cual es el material complementario que se entrega junto con esta memoria, describiendo el árbol de carpetas y los recursos que hay en cada una de ellas. Este material se puede consultar en el siguiente repositorio público de GitHub, <https://github.com/alber82/tfm-k8s>

- Memoria del trabajo en los formatos PDF, DOCX dentro del directorio documentación/memoria.
- En la carpeta *environment* se encuentran los *scripts* para generar el entorno de Kubernetes junto al resto de componentes que se han usado en este proyecto.
- El código fuente del operador se encuentra en la carpeta *operator*.
- El código fuente del planificador se encuentra en la carpeta *scheduler*
- El código fuente del planificador aleatorio que se ha usado para las evaluaciones se encuentra en la carpeta *random-scheduler*.
- El código del *adapter* de postgresql que se usó en las fases tempranas del proyecto se encuentra en *postgresql-prometheus-adapter*.
- En la carpeta *kafka-test* se encuentran los ficheros para poder generar el contenedor que lanza las pruebas de carga.
- En la carpeta documentación/otros hay varios archivos que se han usado de apoyo para el desarrollo del proyecto.
- Libros y artículos a los que se ha hecho referencia durante la memoria y que se han utilizado como bibliografía. Los cuales podemos encontrar en el fichero documentación/bibliografía.
- Páginas Web que han servido de bibliografía. Las podemos encontrar dentro del fichero documentación/enlaces_web.