eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

Orquestadores para la niebla: planificación descentralizada basada en sistemas multi-agente

Alumno	*Diez Martinez, Ioritz*
Director	*Marcos Muñoz, Marga*
Departamento	*Ingeniería de Sistemas y Automática*
Curso académico	*2018/2019*

Bilbao, 27 de mayo de 2019

LABURPENA

Gaur egun fabrika inteligenteari planteatzen zaizkion erronkak industria 4.0 kontzeptuaren barnean, ondokoak hartzen ditu: produktuaren kalitatearen hobekuntza, baita ekoizpen fabriken errendimenduaren optimizazioa ere. Helburu hauek lortzeko, produkzioaren datu masiboen arazoari aurre egiteko, irtenbide bezala “Fog computing” aurkezten da, irregulartasunak aurkitzeko eta bizkortasunez erantzuteko. Aplikazio hauek, testuniguruari sentikorrak diren banatutako aplikazioak bezala ezagutzen dira, testuinguruko aldakatei erantzuten baitiote. Kasu honetan, produkzio aldaketak. Proiektu honetan “cloud” teknologiak erabiltzea proposatzen da, baina aplikazio hauen eragikonrtasunaren baldintzetara egokituz. “Cloud” testuinguruetan, aplikazioak osagai arin birtualizatu edota era zentralizatuan antolatutako edukiontzi bezala exekutatzen dira.

Proiektu honetan antolamendu plataforma baten luzapena proposatzen da. Bertan, bere exekuzioaren planifikazioa negoziazio bitartez emoten da multi agente sistema (MAS) batean oinarritua. “Cloud” plataformen efizientzia, kudeaketaren, hedapenaren eta edukiontzien eskaladoaren automatizazioari dagokionez, erabaki-hartze deszentralizatuari eta MAS sistemen autonomiari lotuta, etorkizunean fabrikazioren berkonfigurazio efizienteak ekinarazteari oinarri sendoak jartzen dizkio, produkzio plantan arazoak detektatzen direnean. Memoria atalean plataformen betekizunen analisi bat aurkezten da, existitzen diren plataformen artean aukeratu ahal izateko. Aukeratutako plataforma describatzen da eta arkitektura luzapenaren espezifikazio teknikoa aurkezten da. Garatutako proposamenaren errendimendua aztertzeko, Bilboko Ingenieritza eskolako GCIS ikerketa taldeak garatutato Fog plataforma errepresentatzen duen prozesadore cluster batean proba batzuk egiten dita.

RESUMEN

Hoy en día los retos que se plantean a la fábrica inteligente dentro del concepto industria 4.0 incluyen, la mejora de la calidad del producto, así como, la optimización del rendimiento de las plantas de producción. Para conseguir estos objetivos, la tecnología “Fog computing” se presenta como una potencial solución a la problemática de procesamiento de datos masivos de producción para detectar anomalías y actuar de forma temprana. Estas aplicaciones se conocen como aplicaciones distribuidas sensibles al contexto, ya que reaccionan a cambios en su entorno. En este caso, de producción. En este proyecto se propone utilizar tecnologías “cloud” pero adaptándolas a los requisitos de eficiencia de estas aplicaciones. En los entornos “cloud” las aplicaciones se ejecutan como componentes virtualizados ligeros o contenedores orquestados de forma centralizada.

En este proyecto se propone la extensión de la arquitectura de una de las plataformas de orquestación de contenedores existentes, que permite una planificación de su ejecución basada en negociación en un sistema multi agente (MAS). La eficiencia de la plataforma “cloud” en cuanto a la automatización de la gestión, despliegue y escalado de contenedores, unido a la toma de decisiones descentralizada y autonomía de los sistemas MAS, forman una base robusta para abordar en un futuro aplicaciones de fabricación reconfigurables y eficientes, cuando se detecten anomalías en la planta de producción. En la memoria se realiza el análisis de requisitos como base para la elección de plataforma entre las alternativas existentes. Se describe la plataforma escogida y se presenta la especificación técnica de la extensión de arquitectura. Para analizar el rendimiento de la propuesta desarrollada, se realizan unas pruebas sobre un clúster de procesadores que representa la plataforma Fog, que ha sido desarrollado en el grupo de investigación GCIS de la Escuela de Ingeniería de Bilbao.

ABSTRACT

Nowadays the challenges that arise in the intelligent factory in the concept of the industry 4.0 include, the improvement of the quality of the product, as well as, the optimization of the performance of the production plants. The technology of fog computation is presented as a potential solution to the problem of mass production data processing to detect anomalies and act early. These applications are known as distributed context-aware applications, and they react to changes in their environment. In this case, manufacturing. In this project it is proposed to use "cloud" technologies but adapting them to the efficiency requirements of these applications. In cloud environments applications are executed as virtualized components or containers orchestrated in a centralized way.

In this project we propose the extension of the architecture of one of the orchestration platforms of the media, which allows scheduling based on the negotiation in a multi agent (MAS) system. The efficiency of the cloud platform in terms of the automation of container management, deployment and scaling, together with decentralized decision making and the autonomy of the MAS systems, form a robust basis for approaching in a near future reconfigurable and efficient manufacturing applications, when anomalies are detected in the production plant. In the memory, it´s made an analysis of the requirements as a basis for choosing the platform among the existing alternatives. The selected platform is described and the technical specification of the extension of the architecture is presented. To analyze the performance of the developed proposal, there are performed some tests on a processor cluster that represents the fog platform, which has been developed in the GCIS research group of the Bilbao School of Engineering.

1 Tabla de contenido

Índice de acrónimos

GCIS: Grupo de Control e Integración de Sistemas.
MAS: Multi-agent system.
iIoT: Industrial Internet of Things.
VM: Virtual Machine.
K8s: Kubernetes.
REST: Representational State Transfer.
CRUD: Create, Read, Update and Delete.
ETCD: Directorio /etc de unix distribuido.
CIDR: Classless Inter-Domain Routing.
API: Application Programming Interface.
GCE: Google Compute Engine.
NAT: Network Address Translation

Índice de Figuras

Índice de tablas

Índice de gráficas

2 Introducción

2.1 CONTEXTO

Con la evolución de la industria de la computación y los grandes avances en las redes de comunicación, la sociedad de hoy en día y en particular su industria, ha experimentado en los últimos años una revolución a la que en 2011 acuñaron con el nombre Industria 4.0 (Kagermann, Wolf-Dieter, & Wolfgang, 2011) . Es claro, que la conectividad está cambiando el entorno social-económico a todos los niveles, y un ejemplo es el doméstico, donde los electrodomésticos se vuelven inteligentes y se conectan a internet siendo posible su gestión remota. A estos elementos se les conoce como dispositivos inteligentes, y se relacionan con el concepto de Internet of Things o Internet de las Cosas (IoT) presentado en 2002 (Huvio, Grönvall, & Främling, 2002) . La industria de hoy en día y sus procesos productivos han tenido que adaptarse a estos cambios, desarrollando una transformación hacía el mundo digital (Zhou, Liu, & Zhou, 2016) . Es por esto que la industria 4.0 se alza bajo 9 pilares tecnológicos que convierten a las fábricas tradicionales en fábricas inteligentes: 1)Big Data y análisis de datos, 2)robots autónomos, 3)simulación (gemelo digital), 4)sistemas para la integración vertical y horizontal, 5)iIot, 6)ciberseguridad, 7)cloud computing, 8)fabricación aditiva y 9)realidad aumentada. Todos estos pilares pueden hacer realidad la fabricación personalizada.

El alto rendimiento, grandes capacidades de procesamiento y almacenamiento, y los bajos costos, hacen del cloud computing una tecnología aplicable en gran variedad de aplicaciones. No obstante, en la producción de hoy en día en la que los plazos de entrega son cada vez más cortos, y los lotes a fabricar más pequeños y personalizados, la alta latencia y ciertos problemas de seguridad pueden hacer que el cloud computing resulte un tanto arriesgado. Como solución a los problemas de alta latencia del cloud, en 2014 cisco propuso el concepto de fog computing (Khakimov, Muthanna, & Muthanna, 2018). En este aspecto, el cloud y el fog están conectados, y como ocurre en la realidad, las nubes quedan más lejos de la tierra que la niebla, haciendo una analogía entre la tierra y el entorno productivo que ofrecen las fábricas a nivel de planta. Luego, lo que permite la niebla es bajar las funcionalidades de la nube a la producción mediante nodos conectados a dispositivos físicos (ver Figura 1).

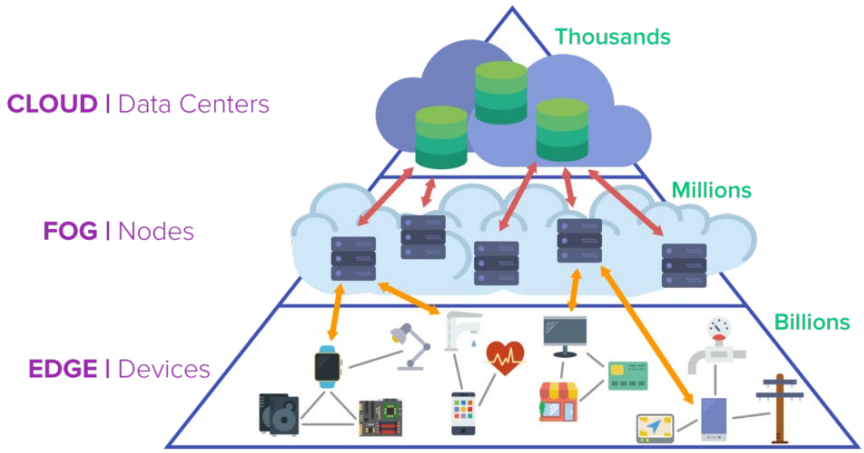


Figura 1 Cloud,Fog y dispositivos

La menor cantidad de datos de transmitidos de la niebla por ser un entorno controlado frente a los centros de procesamiento de datos masivos de la nube (Bonomi, Milito, Zhu, & Addepalli, 2012), permite acelerar una respuesta a los posibles cambios en la producción, desde un fallo en alguna máquina hasta un cambio de pedido (debido a la gran configurabilidad de los productos ofrecidos hoy en día) (Leitão, 2009) . Estas aplicaciones que gobiernan los procesos de fabricación, se conocen como aplicaciones distribuidas sensibles al contexto. Las aplicaciones sensibles al contexto adaptan automáticamente su comportamiento y configuración, dependiendo de las condiciones del entorno y de las preferencias del usuario (Loayza, Proaño, & Camacho, 2013) .

La respuesta de estas aplicaciones a los estímulos externos detectados debe ser rápida. Es por esto, que la ligereza de las aplicaciones se perfila como un parámetro de suma transcendencia. En este aspecto, la ejecución de estas aplicaciones como componentes virtualizados ligeros se presenta como una propuesta conveniente. Concretamente, la tecnología de los contenedores permite una virtualización a nivel de sistema operativo, en lugar de virtualizaciones plataforma (como el de las máquinas virtuales). Además, se pueden crear imágenes de contenedores inmutables en el momento de compilación/ lanzamiento en lugar de en la implementación, ya que no es necesario que cada aplicación esté vinculada al entorno de infraestructura de producción (Kubernetes, 2019i). Con la ventaja de que son virtualizaciones ligeras y portátiles, que pueden llegar a permitir reconfiguraciones dinámicas de estas aplicaciones.

Para Poder realizar reconfiguraciones dinámicas de las aplicaciones (Wang, Wan, Li, & Zhang, 2016), los contenedores requieren de un ente que los gestione de forma eficiente. Desde un correcto escalado, disponibilidad mediante réplicas, hasta despliegues de las aplicaciones. Es por ello que surgen plataformas de gestión automatizada de contenedores, conocidas con el nombre de orquestadores. Sin embargo, se ha observado que ciertos orquestadores tienden a realizar procesos como el despliegue de contenedores de una forma centralizada, que aunque aporta robustez a la plataforma, puede llegar a saturar ese elemento central de toma de decisiones.

Este trabajo se centra en tratar de descentralizar el proceso de despliegue de contenedores que realizan los orquestadores, mediante la ventaja que ofrecen los sistemas multi-agentes en cuanto a la simplificación de la complejidad en tareas distribuidas (GARLAN & SHAW, 2010).

2.2 OBJETIVOS Y ALCANCE.

Se abre este apartado mencionando que los objetivos y alcance del proyecto, han venido marcados por un objetivo que vas más allá del proyecto, analizar la integración de orquestadores de contenedores en la plataforma MAS que el grupo GCIS del Departamento de Ingeniería de Sistemas y Automática de la EHU ha desarrollado.

El objetivo principal de este proyecto es por tanto, el diseño de un proceso de despliegue descentralizado de un orquestador de contenedores, mediante una negociación basada en sistemas multi-agentes (MAS).

Para Poder alcanzar este objetivo, se han de dar una serie de pasos:

- **Conocer la arquitectura y funcionamiento de un orquestador de contenedores**: para proponer un diseño, se ha de conocer los elementos que forman la arquitectura del orquestador, las interacciones entre ellos y el funcionamiento de algunos de ellos relacionados con el despliegue.
- **Configurar un clúster en el que desplegar un orquestador de contenedores**: se han de conocer todas las herramientas necesarias para implantar un orquestador sobre un HW concreto. Herramientas como:
    - **Runtime de los contenedores**.
    - **Creación y carga en el sistema de imágenes de contenedores**.
    - **Conocimiento del API del orquestador para Poder configurar el clúster**.
- **Proponer la "agentificación" de componentes del orquestador de contenedores involucrado en el despliegue**.

Después de identificar los requisitos de integración orquestador-MAS, se analizan las alternativas de orquestadores existentes, seleccionando la más conveniente. Tras esto, se propone un diseño de despliegue distribuido basado en MAS. Además, realiza una implementación del diseño sobre un clúster en el que se ha configurado el orquestador seleccionado. Finalmente, para analizar el funcionamiento del diseño, se realizan una serie de pruebas que determinan su rendimiento, en cuanto a funcionalidad y requisitos temporales.

3 ANÁLISIS Y SELECCIÓN DEL ORQUESTADOR

3.1 Identificación y análisis de requisitos de los orquestadores

Se ha observado que los orquestadores más populares en el mercado, tienden a descentralizar el proceso de despliegue de sus aplicaciones. No obstante, este proceso no se da de forma totalmente distribuida, ya que es el orquestador el que consulta a todos los nodos disponibles y toma una decisión de despliegue, pudiendo llegar en ciertos casos a sobrecargar al elemento central. En este trabajo, se propone la distribución del proceso de despliegue mediante una negociación distribuida entre nodos basada en MAS. Para cumplir con este objetivo, se requieren dos herramientas. Por un lado, un orquestador de contenedores. Por el otro, un MAS en el que se basa la decisión de despliegue distribuida.

En cuanto a la selección de la arquitectura MAS, se basa en las características de los agentes de las que cabe destacar: su **autonomía** (se controlan localmente, sin necesidad de entidades externas), **habilidades sociales** (pueden comunicarse con otros agentes para negociar), **racionalidad** (son capaces de sacar conclusiones en función de datos recibidos y valores propios), **movilidad** (son capaces de moverse entre nodos) y **veracidad** (tienen que decir la verdad) (Nwana, 1996). El framework seleccionado es JADE ya que es en el que está basado el Middleware desarrollado por GCIS (Armentia, Gangoiti, Priego, Estévez, & Marcos, 2015).

En el caso de los orquestadores, se han identificado los requisitos de la Tabla 1 como criterios de selección, y se analizan a continuación:

Tabla 1 Requisitos de los contenedores

Código	Descripción
R1	Código abierto
R2	Escalabilidad
R3	Comunidad de desarrolladores
R4	Tolerancia a fallos
R5	Heterogeneidad de contenedores
R6	Configurabilidad del orquestador
R7	Nivel de implementación
R8	Arquitectura
R9	Documentación
R10	Ligereza

3.1.1 R1 Código abierto

Con objeto de que pueda ser modificado para cumplir con la funcionalidad requerida.

3.1.2 R2 Escalabilidad

La capacidad de escalabilidad de un sistema, es un indicador de las posibilidades y límites de utilización. Si el orquestador es escalable, será capaz de gestionar la información a pesar del tamaño del clúster.

3.1.3 R3 Comunidad de desarrolladores

Una comunidad grande y activa, puede facilitar la comprensión del funcionamiento del orquestador permitiendo a los usuarios centrarse en dicho sistema. De forma indirecta, es un indicador de la aceptación.

Hoy en día, gracias a la gran cantidad de intercambio de información que se realiza en la red, analizar las propuestas existentes pueden mejorar un diseño inicial. No es siempre así, pero por probabilidad, en una comunidad grande la cantidad de información recomendable es también mayor.

3.1.4 R4 Tolerancia a fallos

Este proyecto se sitúa dentro de un marco industrial, en el que se pueden dar diversidad de aplicaciones. No obstante, el funcionamiento general de los sistemas en los que se ejecutan estas aplicaciones debe asegurarse continuamente (Rufino, Alam, Ferreira, Rehman, & Tsang, 2017).

Un caso que ilustra la capacidad de disponibilidad del sistema es el caso de un proceso productivo, donde la parada del mismo debido a un fallo hardware o software puede conllevar grandes pérdidas económicas, por no cumplir a tiempo los pedidos. Existen diferentes fallos que pueden provocar el colapso de un sistema, para comprensión del lector, entre otros se encuentra: fallos de nodos, bugs de los procesos del orquestador y actualizaciones.

3.1.5 R5 Heterogeneidad de contenedores

La diversidad de aplicaciones que se pueden dar en la industria, hacen que no exista un tipo de contenedor que resuelva todo tipo de problemas. Para cada situación, se debe realizar un análisis de la idoneidad del tipo de contenedor. Como consecuencia, un orquestador que permita la ejecución de diferentes tipos, refuerza su posición en el mercado respecto a la competencia.

3.1.6 R6 Configurabilidad de la plataforma y R7 Nivel de implementación

Los procesos productivos, pueden requerir diferentes configuraciones de la plataforma. Por ejemplo, la monitorización de la fabricación de un producto en masa, requiere en función de la aplicación industrial que realicen, unos requisitos temporales de plazos cortos. En otro extremo, una aplicación que analice la calidad de un producto, puede no requerir requisitos temporales tan fuertes, pero si unas capacidades de procesamientos más elevadas.

Es por esto que, cuanto más amplio sea el abanico de configuración que ofrece la plataforma, mejor se puede adecuar ésta al proceso. No obstante, esta gran configurabilidad puede hacer que la instalación de un orquestador, sobre un clúster real, resulte demasiado complicada. Como consecuencia, se debe encontrar un equilibrio entre la configurabilidad ofrecida y el nivel de instalación del orquestador.

3.1.7 R8 Arquitectura

Es necesario que el orquestador haya sido diseñado desde su concepción para ser extensible y esto debe reflejarse en su arquitectura.

3.1.8 R9 Documentación

Un requisito que impone este proyecto es que el orquestador seleccionado disponga de una documentación clara, ordenada y rica en ejemplos, que permita al usuario comprenderla de manera rápida. Este es un requisito importante, dado que disminuye el trabajo en su comprensión y será más asequible extenderlo.

Una amplia comunidad colabora con la elaboración de documentos de calidad, por lo que R3 y R9 están relacionados. Además, ofrece una perspectiva desde un punto de vista más elevado de donde se encuentra la plataforma y hacia donde se quiere dirigir.

3.1.9 R10 Ligereza

Este es un requisito particular de este proyecto, pero curiosamente de forma no directa puede influenciar en la selección de un orquestador frente a otro. Esto es así, porque la implementación del diseño realizado, se debía realizar sobre un clúster con ordenadores de capacidad de procesamiento limitada como son las rasperry Pis.

El orquestador seleccionado tiene que ser capaz de ejecutarse tanto en procesadores pequeños como en otros más potentes.

3.2 Análisis de alternativas

En este apartado se introducen las plataformas de orquestación de contenedores de código abierto (R1) más usadas. Para cada una de ellas, se plantea una pequeña presentación de su funcionamiento y características generales haciendo referencia al cumplimiento de los requisitos introducidos, así como a los componentes principales de su arquitectura.

3.2.1 Docker swarm.

Docker swarm es la plataforma de orquestación de contenedores de Docker. Es una herramienta especialmente recomendable para usuarios que ya conozcan previamente Docker, dado que la interfaz es prácticamente la misma. Se gestiona desde una interfaz de línea de comandos llamada Swarm.

Una propiedad favorable de Swarm es que permite crear un nuevo clúster fácilmente (R6 Configurabilidad de la plataforma y R7 Nivel de implementación). Esto se debe a que Docker, fue concebido en sus inicios como una solución de clúster nativa (Hoque, Brito, Willner, Keil, & Magedanz, 2017). El único requisito de Docker para que un nodo se una al clúster (join), es que tenga instalado docker. No obstante, cumple débilmente con R5 Heterogeneidad de contenedores, ya que solo permite la ejecución de contenedores Docker.

Entre las muchas virtudes de Docker, la simplicidad de su API , le ha hecho ganar aceptación en un público que busca la rápida creación de un clúster, con un nivel considerable de pre-configuración. Sin embargo, el tener un Scheduler distribuido no se encuentra entre una de estas virtudes. Docker presenta un Scheduler monolítico, que aunque aporta robustez y velocidad al sistema, lo hace en cierto modo vulnerable si este elemento colapsa (R4 Tolerancia a fallos).

Docker es una herramienta ligera, que permite su ejecución en dispositivos como rasperry Pis, como indica (Hoque et al., 2017) (R10 Ligereza). Se ha demostrado, que Swarm ha sido capaz de gestionar hasta 5000 nodos, demostrando así buenos niveles de escalabilidad (R2 Escalabilidad).

En la actualidad Docker trabaja estrechamente con Google, compañía con gran respaldo que ha afianzado la posición de Docker en el mercado, con una comunidad amplia (R3 Comunidad). Además, el contenido de los documentos (R9 Documentación) presentes en su web (Swarm features Docker, 2019), junto con la gran actividad entorno a la plataforma, ha facilitado la identificación de los principales componentes de su arquitectura (Isaac Eldridge, 2017), que se representan en la Figura 2

3.2.1.1 Componentes principales de la arquitectura de Swarm :

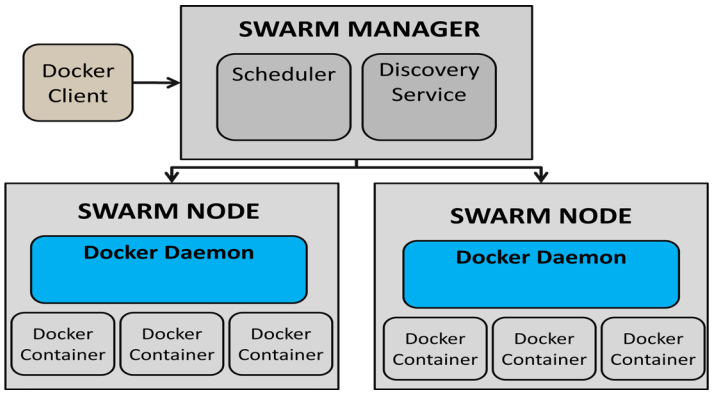


Figura 2 Componentes principales de la arquitectura Swarm (Rajdeep Dua, 2017)

3.2.1.1.1 Swarm

El sistema Swarm consiste en un clúster formado por varios nodos trabajadores (swarm nodes) que pueden ser máquinas físicas o virtuales, con al menos un nodo maestro (swarm manager)

3.2.1.1.2 Service

Un servicio representa la función que debe realizar un nodo o un manager en el clúster, definido por ún administrador Swarm. Además, define la imagen del contenedor y los comandos que se van a ejecutar en cada contenedor. También se pueden especificar ciertos parámetros de configuración en la definición de estos servicios, por ejemplo relativos a las réplicas.

3.2.1.1.3 Manager node

Cuando se despliega una aplicación en Swarm, este componente se encarga de asignar las tareas que cada nodo trabajador debe realizar y gestiona el estado general del clúster al que pertenece.

3.2.1.1.4 Worker nodes

Estos son los nodos que ejecutan la funcionalidad de las aplicaciones que el manager ha desplegado. Cada nodo implementa un agente que informa al manager sobre el estado de las tareas que éste le ha encomendado. Así, el manager monitoriza los servicios y tareas que se desarrollan en el clúster.

3.2.1.1.5 Task

Las tareas son contenedores Dockers que ejecutan los comandos definidos en los servicios. En un despliegue el manager asocia tareas y nodos.

3.2.2 K8s.

Kubernetes es un orquestador de contenedores creado por Google, como evolución de la plataforma Borg y se encarga de automatizar la implementación, escalado y gestión de aplicaciones a través de grupos de hosts (físicos o máquinas virtuales)(Shah, Piro, Grieco, & Boggia, n.d.).

Actualmente, es la herramienta más usada en el mercado y con una mayor comunidad (R3 Comunidad). Además, es el orquestador con más actividad en github, lo que ha permite comprender el funcionamiento del orquestador y resolver múltiples dudas. Además, cuenta con una vasta documentación (R9 Documentación). No obstante, la organización del contenido de la web es mejorable y Podría facilitar aún más el trabajo a sus usuarios.

Se han observado trabajos de k8s sobre host con procesamiento limitado como el caso de k3s (Rancher labs, 2019), donde se comprueba la ligereza del orquestador (R10 Ligereza).

Uno de los éxitos de kubernetes es su arquitectura (representada en la Figura 3) (R8 Arquitectura) modular y distribuida, basada en plugins. Esta, permite al usuario de kubernetes, una gran configurabilidad (R6 Configurabilidad de la plataforma y R7 Nivel de implementación), pudiendo adecuar el orquestador a las características propias de sus aplicaciones y de los hosts que forman el clúster. Con todo, esta gran configurabilidad incrementa la complejidad de la creación de un nuevo clúster (R6 Configurabilidad de la plataforma y R7 Nivel de implementación7). No obstante, el gran número de casos de éxito registrados en la web, permite al usuario simplificar esta posible barrera.

Como se indica en la web (Kubernetes, 2019f), kubernetes es un orquestador capaz de trabajar con hasta 5000 nodos sin sobrecargar de forma desmedida el sistema, pudiendo ejecutar contenedores de diferentes tipos (R5 Heterogeneidad de contenedores). El despliegue en kubernetes se realiza de forma centralizada y declarativa, registrándolo en el sistema en formato JSON. La unidad de despliegue en kubernetes es el Pod (que aloja los contenedores) y el scheduler planifica estos Pods en nodos teniendo en cuenta el estado compartido del sistema.

Como concluye Bellavista & Zanni (2017), los mecanismos de recuperación automática (self-healing), reinicio automático (auto restarting) y replicación y replanificación (rescheduling) de Kubernetes lo hacen más robusto y adecuado para aplicaciones basadas en contenedores.

3.2.2.1 Componentes principales de la arquitectura de k8s:

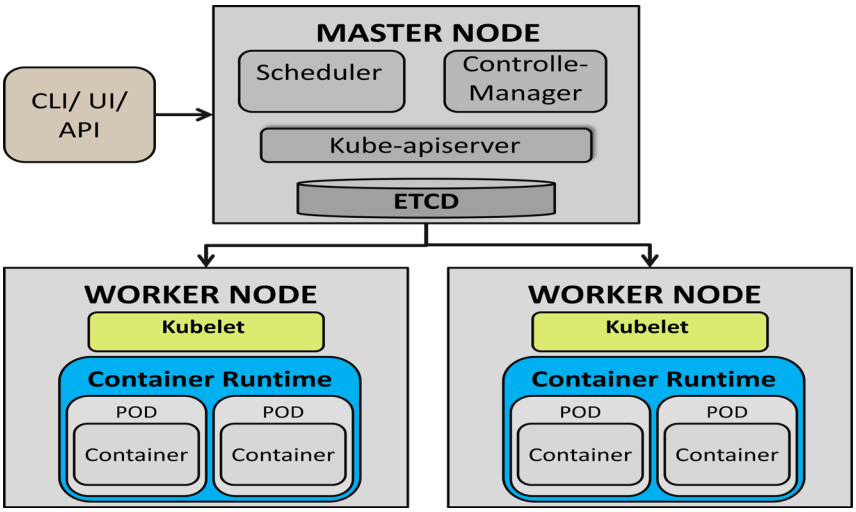


Figura 3 Componentes principales de la arquitectura k8s

3.2.2.1.1 Clúster

Un clúster es un conjunto de nodos con al menos un nodo maestro y varios nodos trabajadores que pueden ser máquinas virtuales o físicas.

3.2.2.1.2 Master

El maestro gestiona el despliegue de instancias de aplicaciones en los nodos, y ofrece una serie de funcionalidades como son el despliegue de aplicaciones en nodos y el control de sus réplicas, entre otros, a través de su servido API (apiserver).

3.2.2.1.3 Kubelet

Cada nodo Kubernetes ejecuta un agente llamado kubelet, que es responsable de gestionar el estado del nodo siguiendo las instrucciones del plano de control. La comunicación entre los nodos trabajadores y el plano de control se hace desde el kubelet hasta el apiserver.

3.2.2.1.4 Pods

Un Pod constituye la unidad de ejecución básica de k8s, y consta de uno o más contenedores con capacidad de compartir recursos. Además, a cada Pod se le asigna una dirección IP única dentro del clúster, lo que permite que la aplicación utilice puertos sin conflicto.

3.2.3 Mesos-Marathon.

Mesos proporciona gestión del clúster y sobre él se pueden construir diferentes entornos como por ejemplo, Marathon que es una plataforma de orquestación de contenedores. Se ocupa de proporcionar el descubrimiento de servicos, balanceo de carga, gestión de los recursos del clúster y una API para gestionar las cargas de trabajo.

La plataforma Mesos no fue concebida inicialmente para trabajar como un orquestador, es por eso que además de gestionar aplicaciones en contenedores, puede gestionar también máquinas virtuales (VM). En cuanto a los contenedores, está diseñada para trabajar con Mesos o Docker (R5 Heterogeneidad de contenedores).

Mesos fue concebido como una herramienta para gestionar clústeres ya implantados, y una de sus principales propiedades es su robustez. Ofrece una alta tolerancia a fallos (R4 Tolerancia a fallos), siendo capaz de aislar las partes en fallo del sistema. Sin embargo, la creación de un nuevo clúster desde el inicio, o la adición de nuevos nodos al sistema puede resultar en un proceso tedioso (R6 Configurabilidad de la plataforma y R7 Nivel de implementación). Para que un nodo pueda unirse al sistema, requiere tener instaladas dos herramientas, Mesos y Zookeper, conociéndose este conjunto como Marathon. Además, cada nodo debe contar con un archivo de configuración que permite al nodo maestro identificarlo en el sistema. Como indica la compañía en su web (Mesos, 2019), la plataforma es capaz de gestionar hasta 10000 nodos (R2 Escalabilidad).

Mesos ofrece una API sencilla que permite el despliegue de las aplicaciones, mediante un scheduler centralizado de dos niveles. No obstante, debido al diseño de su arquitectura, la configuración de un scheduler customizado, puede resultar un trabajo no trivial (R6 Configurabilidad de la plataforma y R7 Nivel de implementación).

El orquestador que se presenta tiene unos grandes resultados en cuanto a robustez y seguridad en su ejecución, pero a costa de un gran consumo de memoria. Esto implica que, es lo suficientemente pesado como para no Poder ejecutarse en un ordenador con baja potencia de procesamiento, como las rasperry Pis (R10 Ligereza) (Hoque et al., 2017).

Finalmente, Mesos como plataforma perteneciente a la empresa Apache cuenta con una amplia comunidad, con gran cantidad de documentos y recursos (R9 Documentación,R3 Comunidad) (Shah et al., n.d.).

3.2.3.1 Componentes principales de la arquitectura de Mesos+Marathon:

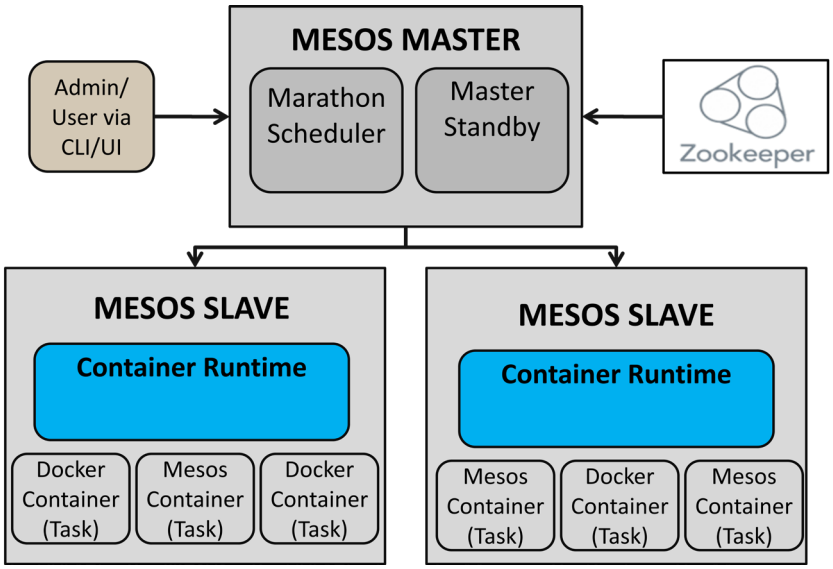


Figura 4 Componentes principales de la arquitectura Swarm (Isaac Eldridge, 2017)

3.2.3.1.1 Master daemon

Parte del maestro que gestiona los agentes que se ejecutan en su nodo. Para crear clústeres a gran escala y alta disponibilidad (una de las mejores cualidades de esta opción), se debe tener un maestro y dos réplicas (standby), y con la herramienta Zookeper (ilustrada en la Figura 4) se implementa un quórum que gestiona el clúster.

3.2.3.1.2 Agent daemon

Agentes que forman parte del maestro y ejecutan tareas ordenadas por el gestor de contenedores.

3.2.3.1.3 Framework

Como Mesos no tiene capacidad de orquestación, Marathon es el framework se superpone a Mesos. Para ello, Marathon procesa la información de los recursos enviada por Mesos, y envía tareas que deben ejecutar los agentes de los nodos esclavos.

3.2.3.1.4 Offer

Mesos reúne la información sobre los recursos y memoria disponible en los nodos a través de sus agentes y envía esta información a Marathon. Así, este conoce la disponibilidad del sistema.

3.2.3.1.5 Task

Las tareas son las unidades básicas de trabajo que ejecutan los nodos esclavos, asignadas por Marathon a través del maestro.

3.2.4 HashiCorp Nomad

Nomad es el framework para la orquestación de contenedores del grupo Hashi. Presenta una gran simplicidad en su arquitectura (R8 Arquitectura), que permite una rápida y fácil formación de nuevos clústeres. Es una herramienta muy flexible que permite controlar, desde diferentes tipos de contenedores (R5 Heterogeneidad de contenedores), pasando por JARS (Java), hasta binarios que se ejecutan como root sin ningún aislamiento.

La simplicidad de su arquitectura, junto a su fácil instalación y su gran pre configuración (R6 Configurabilidad de la plataforma y R7 Nivel de implementación) perfilan a esta herramienta a usuarios que buscan rápidos resultados en la configuración de un clúster. Asimismo, estas propiedades convierten a esta plataforma en ligera (R10 Ligereza).

La modularidad de su arquitectura, presenta una fortaleza según indican en su web, para la alta tolerancia a fallos (Hashicorp, 2019) (R4 Tolerancia a fallos), siendo capaz de aislar las partes dañadas y recuperar datos relevantes afectados, asegurando así un funcionamiento general del sistema adecuado. Se presenta esta arquitectura en la Figura 5.

Precisamente una de las potencias de Nomad, es que permite una rápida gestión de clústeres de gran escala, llegando a controlar incluso 10000 nodos (HashiCorp, 2019) (R2 Escalabilidad). El único requisito que el sistema impone a un nodo para unirse al clúster, es tener Nomad instalado. Sin embargo, el proceso de despliegue sigue siendo un tanto centralizado, ya que su scheduler actúa de forma declarativa (similar a K8s) en función del estado compartido del sistema. Está pensado para planificar servicios que nunca terminan (long live services). Para ello, ordena a todos los nodos que tienen los suficientes recursos y selecciona la ubicación óptima para los trabajos (Job).

Por último, Nomad cuenta con una comunidad similar a la de Marathon, con una buena documentación y gran cantidad de casos de éxito.

3.2.4.1 Componentes principales de la arquitectura de Nomad:

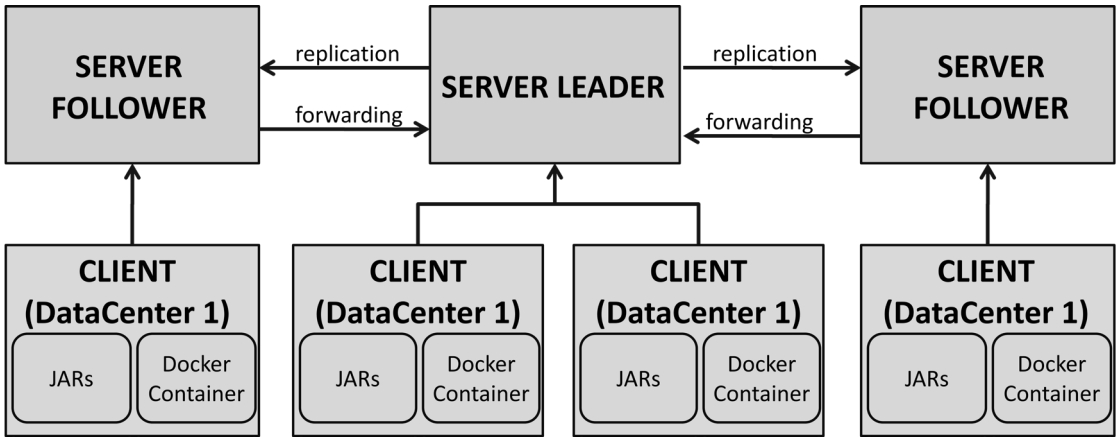


Figura 5 Componentes principales de la arquitectura Nomad, con una región.

3.2.4.1.1 Server

Los servidores, representan el cerebro del clúster. Hay un grupo de servidores por región que gestionan todos los trabajos y clientes, evalúan alternativas y crean asignaciones de tareas. Los servidores replican datos entre sí y realizan una elección de líder para garantizar una alta disponibilidad. Los servidores se federan a través de las regiones para que Nomad tenga un conocimiento global del clúster.

3.2.4.1.2 Client

Un cliente de Nomad es una máquina en la que se pueden ejecutar las tareas. Todos los clientes ejecutan el agente Nomad. El agente es responsable de registrarse en los servidores, estar a la espera de que le asignen, y ejecutar tareas.

3.2.4.1.3 Regions y Datacenters

Nomad modela la infraestructura del sistema con regiones y centros de datos. Las regiones pueden contener múltiples centros de datos. Se asignan ciertos servidores a cada región y se encargan de gestionar el estado y tomar decisiones de programación dentro de esa región. Múltiples regiones pueden pertenecer a la misma federación, pero no se replican los datos entre regiones.

3.2.4.1.4 Task

Las tareas son las unidades básicas de trabajo que ejecutan los clientes, planificadas por los servers.

3.2.4.1.5 Job

Un trabajo es una especificación proporcionada por los usuarios que declaran una carga de trabajo (workload) para Nomad. Es una forma de estado deseado, donde el usuario expresa qué trabajo debe estar ejecutándose, pero no donde debe ejecutarse. La responsabilidad de Nomad es asegurarse de que el estado real del clúster (proporcionado por los agentes de los clientes) coincida con el estado deseado por el usuario. Un trabajo se compone de uno o más grupos de tareas.

3.2.4.1.6 Protocolo Gossip

Protocolo para gestionar los miembros del clúster.

La Tabla 2 resume el análisis realizado de cumplimiento de los requisitos lo que permite tomar la decisión de selección.

3.3 Tabla comparativa: Requisitos vs Alternativas

Tabla 2 Análisis de requisitos

	Orquestadores			
Requisitos	**Docker Swarm**	**Kubernetes**	**Mesos-Marathon**	**HashiCorp Nomad**
R1	OpenSource (ApacheLicense2.0)	OpenSource (ApacheLicense2.0)	OpenSource (ApacheLicense2.0)	OpenSource (ApacheLicense2.0)
R2	Despliegue y escalado rápido incluso con muchos contenedores.	Despliegue y escalado con poca consideración de la velocidad, pero gran número de contenedores.	Gran capacidad de escalabilidad.	Despliegue y escalado con poca consideración de la velocidad, gran número de contenedores.
R3	Grande	Grande	Media	Media
R4	Baja	Alta	Alta	Media
R5	Baja, sólo Docker.	Alta, Docker, AWS, Azure, IBM…	Baja, Mesos y Docker.	Alta, soporta app virtualizadas, contenedorizadas y standalone
R6	Medio y muy sencillo,	Muy alto, y de nivel medio	Baja, y recomendable cuando ya se tiene un clúster.	Baja, pero muy sencillo.
R7	Sencillo	Dificultad alta dada su gran configurabilidad.	Sencillo para clústeres ya creados, sino tedioso.	Sencillo
R8	Media	Media-compleja	Media	Sencilla
R9	Alta	Muy alta	Media	Media-baja
R10	Ligero	Medio-ligero	Pesado	Ligero

3.4 Selección del orquestador

Analizando la Tabla 2 se ha seleccionado k8s por diferentes razones:

Kubernetes es el orquestador con más actividad en la actualidad, con una comunidad grande y muy activa, que ofrece soporte y consigue que evolucione y presenta nuevas versiones estables cada poco tiempo. Además, cuenta con una gran cantidad de ejemplos y casos de éxito en muy diversos ámbitos de aplicación.(R3)

Desde el punto de vista de consumo de recursos, kubernetes no representa la opción más ligera, Swarm y Nomad lo son más, pero es lo suficientemente ligera para ejecutarse en el clúster del que dispone el grupo de investigación GCIS del Departamento de Ingeniería de Sistemas y Automática de la EHU compuesto por cuatro rasperry Pis. (R10)

La configurabilidad de kubernetes decanta la balanza muy a su favor, ya que permite al usuario un alto nivel de customización del orquestador. Así, se puede adecuar en mayor medida al marco de actuación del mismo. Más allá, su arquitectura modular y extensible, basada en plugins, permite ampliar la funcionalidad del orquestador modificando o extendiendo alguno de sus componentes. Esto último, coincide plenamente con el objetivo principal del trabajo. (R6-R8)

Finalmente, los clústeres orquestados con kubernetes resultan más robustos y completos, que por ejemplo los creados con Swarm. Esto es debido a la variedad y cantidad de contenedores que puede gestionar. La distribución de las funcionalidades del orquestador entre los diferentes componentes de la arquitectura, permiten aislar estos componentes en caso de fallo de alguno de ellos, aumentando así la tasa de tolerancia a fallos. (R2, R4, R5)

Las razones expuestas posicionan a k8s como la herramienta más adecuada para cumplir con el objetivo del proyecto.

3.5 Identificación y Análisis de riesgos

En este apartado se introducen los riesgos que pueden darse durante el desarrollo del proyecto. En primera instancia se analizan los problemas que se pueden encontrar en la Fase 1: Formación, Fase 2: Diseño de extensión/modificación de arquitectura, y Fase 3: Implementación del diseño. Tras la implementación del diseño, y para analizar la funcionalidad y los requisitos temporales de la propuesta, se realizaron una serie de pruebas (F4). El código de las fases se refleja en el apartado de METODOLOGÍA.

3.5.1 Riesgos en la extensión, distribución y agentificación del orquestador:

1. En la definición de funciones que tienen que realizar los agentes.
2. En la asociación de funciones-agentes.
3. En la cantidad de agentes necesarios, en la arquitectura del sistema.
4. En las limitaciones de extensibilidad del orquestador.
5. En la integración del MAS y el orquestador (agentificación de un componente del orquestador).

3.5.2 Riesgos durante las pruebas de rendimiento del sistema:

Los riesgos que se pueden dar al hacer mediciones de la respuesta del sistema son muchos:

6. En una mala elección de las variables en las pruebas cuantitativas.
7. En una elección equivocada de la aplicación o aplicaciones empleadas, no siendo suficientemente ricas para analizar la funcionalidad de la implementación.
8. En la medición incorrecta de las pruebas temporales, por uso de equipos no adecuados.
9. En la implementación del diseño propuesto, que no demuestre la potencia estimada.

Estos riesgos mencionados deben tenerse muy en cuenta, ya que implican un resultado de la respuesta del sistema. Son importantes porque en este proyecto se busca que los tiempos en el despliegue bajen y aumente la disponibilidad del sistema, al disminuir la carga del maestro, evitando sus posibles sobrecargas.

3.5.3 Riesgos de gestión del proyecto:

10. En la definición del alcance muy ambicioso del proyecto, ya que además de una fase de ejecución, este proyecto ha tenido una fase importante de investigación.
11. En la planificación temporal.
12. En la inversión de horas del personal, que hace que el presupuesto se dispare. Producidas por la complejidad de.

3.5.3.1 Riesgos habituales:

13. En la pérdida de tiempo en labores que finalmente no ayudaron al desarrollo del proyecto.
14. En la formación no adecuada al desarrollo del proyecto.
15. En la falta de conocimiento de diferentes materias en varios puntos del proyecto.
16. Al tener una gran parte de proceso de investigación, la continúa redefinición de los objetivos del proyecto para encontrar el éxito.

3.5.4 Mapa semántico de probabilidad/Importancia:

Se ilustra un mapa semántico con la probabilidad de cada riesgo y su importancia en el proyecto en la Figura 6.

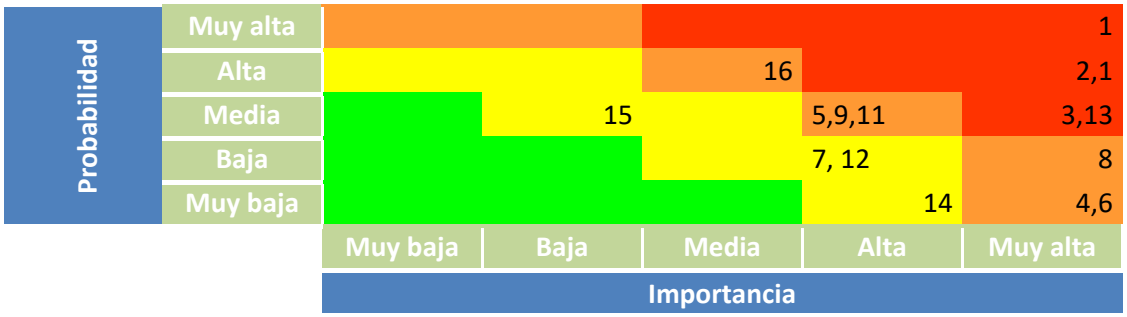


Figura 6 Mapa semántico de probabilidad/Importancia

3.5.5 Plan de contingencia

Una correcta identificación de los riesgos presentes en este proyecto, junto con las acciones que pueden mitigar estos riesgos, es clave para Poder controlar el desarrollo del mismo. A continuación se presentan las acciones propuestas (ver Tabla 3).

Tabla 3 Acciones de contingencia

Riesgos	Descripción de la acción de contingencia
1	Profundización en el funcionamiento del orquestador, llegando a analizar el funcionamiento de ciertos componentes de la arquitectura a nivel de codificación.
2	
3	
4	Una buena definición de los requisitos que se le exigen a un orquestador para Poder alcanzar el objetivo principal de este proyecto, junto con un buen análisis de alternativas existentes, para la selección del orquestador más adecuado.
5	
6	Un claro conocimiento del objetivo del proyecto, facilita la elección de las gráficas representantes del funcionamiento del diseño propuesto.
8	El uso de equipos con baja capacidad de procesamiento, permite obtener mayores diferencias temporales debido a posibles sobrecargas en ciertos componentes del orquestador.
9	El cercano seguimiento de la directora del proyecto mitiga este riesgo, con una rápida detección de posibles fallos de diseño.
10	Un seguimiento exhaustivo del proyecto por parte de la directora, planificando hitos y objetivos parciales alcanzables a corto plazo facilita la visión del alcance global.
11	Definición de hitos y plazos de finalización de objetivos.
13	La consulta de casos de éxito similares al que se desarrollan en este proyecto sobre las tecnologías necesarias para la evolución del mismo, junto con la definición de objetivos alcanzables en un corto plazo.
16	Un seguimiento exhaustivo del proyecto por parte de la directora, planificando hitos y objetivos parciales alcanzables a corto plazo facilita la visión del alcance global.

4 DISEÑO

4.1 K8s Conceptos y Arquitectura

4.1.1 Introducción

Kubernetes es una plataforma de código abierto, extensible y portátil para administrar cargas de trabajo y servicios en contenedores, que facilita tanto la configuración declarativa como la automatización. Tiene un ecosistema grande y de rápido crecimiento. Es un proyecto que presentó google en 2014 como evolución de la anterior plataforma Borg (que no era de código abierto). Kubernetes se basa en una década y media de experiencia que Google tiene al ejecutar cargas de trabajo de producción a gran escala, combinadas con las mejores ideas y prácticas de la comunidad.

Kubernetes proporciona un entorno de gestión **centrado en contenedores**. Organiza la infraestructura de computación, redes y almacenamiento en nombre de las cargas de trabajo de los usuarios. Esto proporciona gran parte de la simplicidad de la Plataforma como un Servicio (PaaS) con la flexibilidad de la Infraestructura como un Servicio (IaaS), y permite la portabilidad entre los proveedores de infraestructura.

K8s es un diminutivo de Kubernetes, viene de sustituir las 8 letras “ubernete” por un 8 y en la literatura que sigue se designará de este modo (Kubernetes, 2019i).

4.1.2 Objetos k8s

Toda información persistente en el sistema k8s se representa como un objeto API. Los objetos de K8s son elementos duraderos, que se usan para representar el estado deseado del sistema. El plano de control de k8s trata de llevar el estado actual del sistema al deseado (el registrado por el usuario a través de los objetos). Estos objetos se utilizan para describir entre otras cosas:

- Qué aplicaciones están ejecutándose y en qué nodos.
- Los recursos disponibles para Poder ejecutar las aplicaciones.
- Las normas que indican el funcionamiento de las aplicaciones.

Los objetos de K8s se crean, modifican y eliminan a través del API de K8s. Sobre estos objetos se realizan operaciones CRUD (Crear, Leer, Actualizar y Eliminar) a través del API que es de tipo REST (con llamadas http). La comunicación con el servidor del API (apiserver) (encargado de modificar los objetos en última instancia) se puede hacer mediante:

- **Kubectl** una interfaz de línea de comandos, donde usuario indica su deseo de una forma declarativa y automatizada (este componente se encarga de hacer las llamadas REST necesarias al apiserver). Este último elemento se encarga de registrar las llamadas recibidas en formato JSON en el etcd. Sin embargo, la opción más cómoda para registrar un estado deseado, es mediante archivos YAML.
- **Librerías clientes** de k8s, con esta opción se puede usar directamente la API.

Los objetos k8s siguen una estrucutra determinada por dos campos anidados, la **especificación** (spec) del objeto y el **estado** (status). Por una parte, la especificación es un campo obligatorio, que describe el estado deseado para el objeto. Es un campo que debe introducir el usuario en registro. Por otra parte, el estado es un campo que actualiza con cierta frecuencia (configurable) el sistema k8s. La función principal del plano de control (que se describe más adelante) es la de leer el estado deseado de los objetos y tratar de que coincida con el actual del objeto.

Un buen ejemplo para entender lo anterior se da cuando el usuario registra un objeto de tipo despliegue en la que quiere que se ejecuten tres réplicas de una aplicación. El sistema inicia ese despliegue y monitoriza su actividad, pero si en algún momento cambia el estado de alguna de esas réplicas (p.e. porque ha caído el nodo en el que se ejecutaban) el plano de control de k8s responde al cambio entre estado deseado y actual. En este caso inicia una nueva instancia de la aplicación para mantener el número de réplicas.

Además de los campos descritos anteriormente, conviene registrar información básica del objeto en un campo denominado **metadata**, que contiene información como el nombre, UID (Unique IDentifier)...

La Figura 7 representa el despliegue de una aplicación, donde se puede observar que el despliegue representa una aplicación con la etiqueta nginx, que a su vez contiene un contenedor con 2 réplicas (3 contenedores en total) con nombre nginx también. Estos contenedores ejecutan una imagen con la versión 1.7.9, dejando el puerto 80 de cada contenedor abierto para comunicaciones.

controllers/nginx-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Figura 7 Ejemplo de estructura de un objeto de despliegue

4.1.2.1 Objetos básicos

Con la misma estructura presentada, existen varios objetos básicos con relevante importancia en el sistema k8s. En este apartado se introducen cuatro concretamente, siendo uno de ellos de mayor interés para el desarrollo del este proyecto, el **Pod**.

4.1.2.1.1 POD

Se presenta el Pod como un envoltorio en el que se ejecuta uno o más contenedores. POD en castellano significa vaina. En la Figura 8 se ilustra la idea presentada.

Figura 8 POD

Un Pod puede crear o desplegar, siendo el despliegue un objeto de gran interés en este proyecto. Desde el punto de vista de la programación, representa la instancia de una aplicación en el sistema K8s, el cual puede contener uno o más contenedores. No obstante, k8s recomienda seguir la filosofía de un contenedor por Pod.

Los Pods tienen una IP única y especificaciones de cómo se tienen que ejecutar sus contenedores. Además, pueden tener unidad/es de almacenamiento compartido entre los contenedores. Se pueden usar de dos formas:

- Un contenedor por Pod: es el modelo más común, y el que defiende k8s.
- Varios contenedores por Pod: la instancia de una aplicación (un Pod) puede necesitar la existencia de varios contenedores relacionados por medio de recursos compartidos, para una mejor ejecución. El Pod agrupa este conjunto de recursos y contenedores como una única entidad, formando unidades de servicio. Así, los contenedores dentro de un mismo Pod pueden compartir recursos, comunicarse entre ellos y coordinarse para cuando se tienen que terminar.

El ciclo de vida de un POD comprende desde el momento de su creación, pasando por asignación de un UID, planificación (Schedule), ejecución y hasta su terminación terminación.

Tras su creación un Pod pasa por distintas fases, que se representan en un campo conocido como **fase**, dentro del campo status del objeto. Este campo se actualiza cada cierto tiempo por el plano de control de k8s. La fase es un resumen de alto nivel que indica dónde se encuentra (a quien lo consulte) el Pod dentro de su ciclo de vida, pudiendo pasar por las siguientes:

- **Pending**: es la fase inicial de un Pod aceptado en el sistema. Sin embargo, aún no se han creado las imágenes de los contenedores que aloja.

- **Running**: se encuentra en esta fase una vez que ha sido asociado al nodo en el que se tiene que ejecutar, se crean las imágenes de sus contenedores y se ejecuta al menos una.
- **Succeeded**: un Pod se encuentra en esta fase cuando todos sus contenedores han ejecutado su funcionalidad con éxito, y no necesitan ser reiniciados.
- **Failed**: se da cuando en ejecución al menos uno de sus Pods ha fallado.
- **Unknown**: cuando por cualquier razón el sistema no puede obtener el estado de un Pod. Es una fase que se suele dar generalmente por errores de comunicación.

La Figura 9 represena las transiciones posibles entre estos estados.

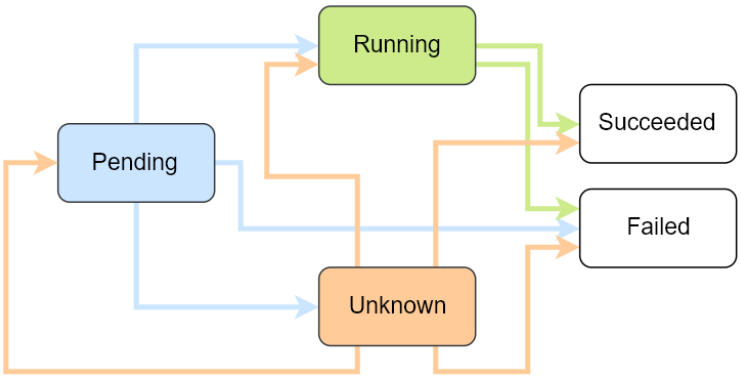


Figura 9 Fases del ciclo de vida de un Pod.

Como objetos que agrupan contenedores, el contenido de un Pod se planifica y ejecuta compartiendo los recursos de un mismo nodo. Así, todo lo que envuelva el POD tiene la misma vida que él. Si tras la muerte de un nodo en el que se ejecutaba un POD, se vuelve a levantar, sus elementos no reviven sino que se crean nuevos con el mismo nombre, pero con un nuevo UID.

Como se ha presentado, los Pods alojan contenedores y **Docker** es una herramienta (entre otras alternativas) que gestiona el runtime de los contenedores. Se puede establecer una analogía entre la gestión de Docker y los contenedores, con la de k8s y los Pods. Una vez se crea la imagen de un contenedor en un Pod, el contenedor puede pasar por tres diferentes estados:

- **Waiting**: tras la creación del contenedor, es el estado que adquiere por defecto. El plano de control actualiza junto con el estado un campo llamado **reason**, que describe la razón por la que se encuentra en este estado.
- **Running**: es el estado en el que el contenedor ejecuta su funcionalidad sin fallos. El plano de control registra el **instante temporal** en que el contenedor conmuta a este estado.
- **Terminated**: pasa a este estado cuando termina su ejecución, ya sea porque ha ejecutado completamente su funcionalidad o porque ha fallado. No obstante, tras entrar en este estado se registra el **instante temporal**, junto con un **código** que indica la razón por la que ha conmutado a este estado.

La unión lógica que hacen los Pods de los contenedores que se ejecutan en él, facilita el despliegue y gestión de las aplicaciones (que pueden estar formadas por uno o más contenedores). Esto es así porque eleva la gestión de una aplicación, del nivel de contenedor a nivel de Pod (ver Figura 10).

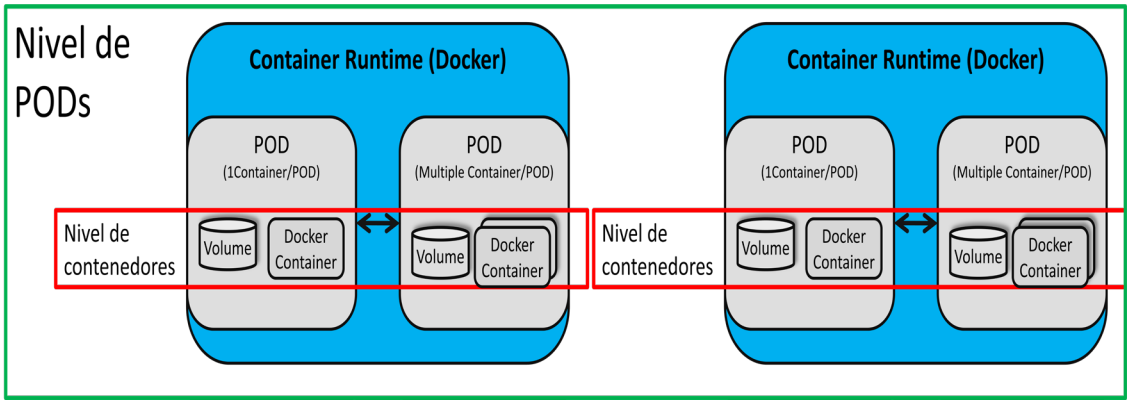


Figura 10 Nivel de Pods

Los Pods tienen dos tipos de recursos compartidos; de **red** y de **almacenamiento**.

El Pod representa la mínima unidad de gestión y todos los contenedores que residen en él comparten el mismo interfaz de red, por lo que pueden comunicarse entre entre ellos utilizando el host local.

Los datos que se almacenan en un contenedor son temporales. En consecuencia, si un contenedor falla, se crea uno nuevo, suponiendo la pérdida del estado. Esto puede gestionarse si es necesario restaurando los datos si se utilizan volúmenes como se explica a continuación.

4.1.2.1.2 VOLUME

En un Pod que contiene uno o más contenedores es necesario mantener el estado aunque se produzca un fallo, se utiliza el concepto de volumen (volumen). Éste, consiste en una zona de memoria compartida por los contenedores de un mismo Pod. Cómo se origina ese directorio, el medio que lo respalda y su contenido están determinados por el tipo de volumen particular utilizado (azureDisk, azureFile, cephfs, configMap, csi…). Un volumen tiene la misma vida que un Pod, sobreviviendo los datos que almacena a posibles reinicios de contendores, y sería posible restaurar un Pod con los datos que tenía antes del fallo.

4.1.2.1.3 SERVICE

Como se ha comentado anteriormente, k8s asegura la disponibilidad de un Pod. Es decir, si fallan, los recupera creando nuevos Pods con la misma funcionalidad, por lo que pueden tener asignadas nuevas IPs. Existen casos en los que varios Pods (llamados **backends**) que ofrecen funcionalidad a otros que la consumen (llamados **frontends**). Pero si uno de esos backend falla, los frontends Podrían no localizarlo. Para solucionar esto, surge el concepto de servicio (**service**).

Un servicio Kubernetes es un objeto que une de manera lógica un conjunto de Pods y una política mediante la cual acceder a ellos.

Para ilustrar el concepto de servicio supongamos una aplicación que procesa imágenes que se ejecuta con 3 réplicas. Para poder procesar estas imágenes, ésta aplicación (como frontend) requiere el uso de varios contenedores (que actúan como backends) que ejecutan la funcionalidad que permite procesar las imágenes. Gracias al concepto de servicio, los frontends no tienen por qué hacer un seguimiento de los backends (por si alguno de ellos hubiese fallado) que requieren para ejecutar la funcionalidad que demandan, sino que hacen un seguimiento del servicio, y es el servicio el encargado de indicar con qué backends se tienen que comunicar.

4.1.2.1.4 NAMESPACE

K8s admite varios clústeres virtuales respaldados por el mismo clúster físico. Estos clústeres virtuales, se denominan espacios de nombres (namespace). No se adentra más en este tema porque k8s recomienda su uso en entornos con muchos usuarios distribuidos en varios equipos o proyectos, quedando fuera del alcance de este proyecto.

4.1.2.2 Controllers

K8s contiene una serie de objetos con abstracciones de nivel superior denominadas controladores (**controllers**). Los controladores se basan en los objetos básicos y proporcionan funciones adicionales. A continuación, se introducen dos controladores de relevancia en este proyecto.

4.1.2.2.1 DEPLOYMENT

Un controlador se encarga de: crear y gestionar Pods, gestionar las réplicas y los despliegues, y ofrecer habilidades de regeneración (self-healing). Un controlador que se presentará en mayor profundidad dado su interés, es el **Deployment controller**. Encargado de el despliegue de Pods.

Los controladores crean Pods gracias a las plantillas (templates), objetos de configuración de los Pods. Implementando una plantilla de un tipo, se puede volver a crear un nuevo Pod idéntico.

Un concepto importante en el diseño de k8s que le distingue de otras opciones como; Apache Hadoop Yarn p.e., es que cada tipo de proceso (creación de réplicas, despliegue de réplicas, monitorización de salud,…) es controlado por un solo controlador. Así por ejemplo, cualquier proceso de despliegue que pueda haber en el sistema es controlado por el Deployment controller. El masetro k8s gestiona de forma centralizada al conjunto de controladores, que aunque aporta robustez al sistema, puede resultar en una debilidad (por una posible sobresaturación del maestro).

4.1.2.2.2 REPLICASET

El encargado de gestionar las réplicas es un controlador llamado **ReplicaSet**. El propósito de un ReplicaSet es mantener el conjunto deseado de réplicas en ejecución en un momento dado. Es decir, su función es garantizar la disponibilidad de un número específico de Pods idénticos.

4.1.3 Plano de control de k8s

El plano de control mantiene un registro de todos los objetos k8s presentes en el sistema y ejecuta bucles de control para administrar el estado de esos objetos. Cada cierto tiempo, los bucles de control responderán a los cambios detectados en el clúster y trabajarán para hacer que el estado real de todos los objetos coincida con el estado deseado registrado por el usuario, gracias al PLEG (Pod Lifecycle Event Generator). El plano de control realiza una variedad de tareas de forma automática, como iniciar o reiniciar contenedores, escalar el número de réplicas de una aplicación determinada entre otras. El plano de control lo compone una colección de procesos que se ejecutan en el clúster.

Por un lado, el nodo que actúa como maestro. Aquí, es donde se registra el estado deseado del clúster, que k8s es responsable de mantener. Dentro del nodo maestro se ejecutan cuatro procesos que corresponden a los componentes; **kube-apiserver**, **kube-controller-manager**, **cloud-controller-manager** y **kube-scheduler** . Además, el maestro puede ser replicado para asegurar la disponibilidad del mismo.

Por otro lado, en el resto de nodos del clúster se ejecutan dos procesos que corresponden a los componentes de la arquitectura **kubelet** y **kube-proxy**. Los nodos de un clúster son las máquinas (máquinas virtuales, servidores físicos, etc.) que ejecutan sus aplicaciones y flujos de trabajo en la nube. La Figura 11muestra los elementos del plano de control de k8s.

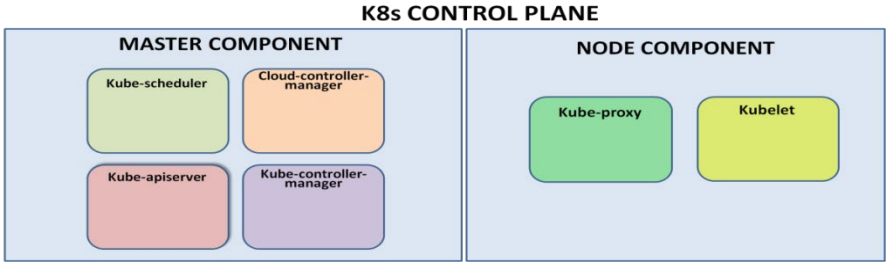


Figura 11 Plano de control de k8s

4.2 Arquitectura de k8s

La Figura 12muestra la arquitectura de componentes de K8s. A continuación, se presentan sus funciones principales y ciertas interacciones entre elementos que se consideran relevantes.

La arquitectura está formada por componentes conectables, con la capacidad de usar planificadores, controladores, sistemas de almacenamiento y mecanismos de distribución alternativos y configurables.

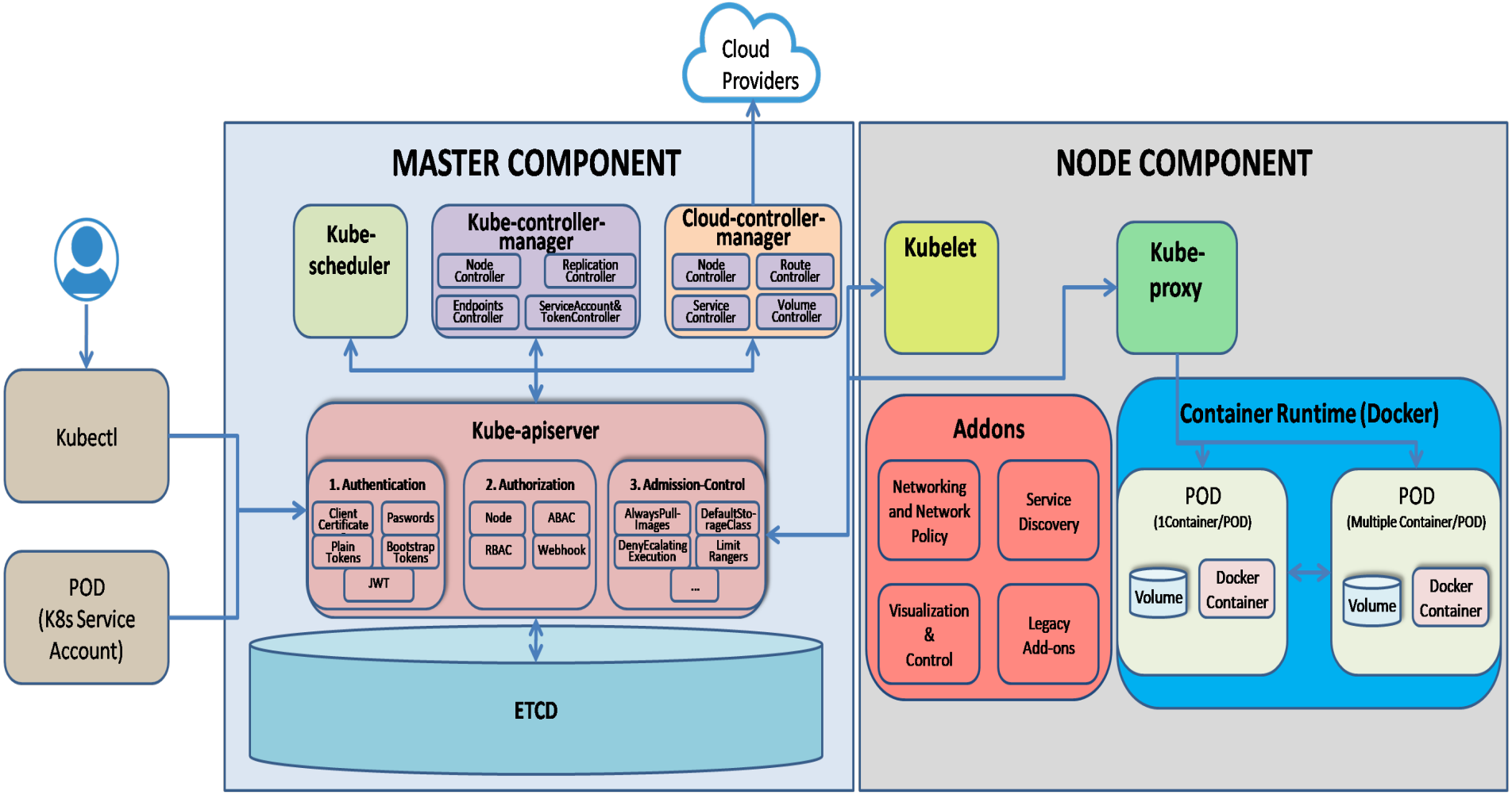


Figura 12 Arquitectura k8s

4.2.1 Componentes de la arquitectura

4.2.1.1 Componentes maestros

El maestro, como se introdujo previamente, forma parte del plano de control de un clúster de k8s. Es el elemento central de toma de decisiones, y como tal detecta y responde a eventos del sistema. Los componentes del maestro (y sus réplicas) pueden ejecutarse en cualquier nodo. No obstante, se recomienda desplegar a cada maestro y los componentes que lo forman en un mismo nodo independiente del resto de nodos trabajadores del clúster. Además, maestro ofrece el API de K8s que permite la interacción usuario-clúster.

4.2.1.1.1 Kube-apiserver

Kube-apiserver es lo que otros elementos de la arquitectura ”ven” del plano de control, ya que ofrece el API de K8s. Tanto el usuario como los controladores, se comunican con el clúster a través de él (**frontend** al clúster, y a su vez al plano de control), como se representa en la Figura 13. Los usuarios externos tienen que Poder acceder al apiserver, pero no así a los contenedores que hay en los nodos. Por esto, se dice que este componente actúa como un **Gateway** hacia el clúster.

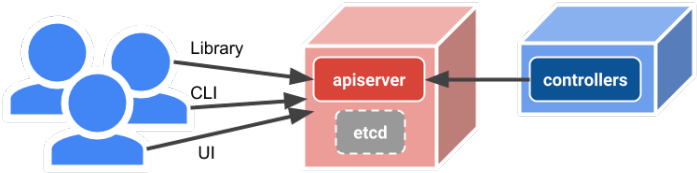


Figura 13 Diferentes formas de comunicarse con el apiserver. (Kubernetes, n.d.-b)

Las operaciones y comunicaciones entre componentes, y los comandos externos (de usuarios), son en realidad llamadas al API gestionadas por el **apiserver**. Toda llamada al API hace referencia a un objeto de la plataforma, y el apiserver implementa funcionalidades comunes para todos estos objetos. Además, valida y configura datos para los objetos del API como Pods, servicios, replication controllers....

El apiserver ejecuta toda la lógica en componentes separados (plug-ins). Cuando el apiserver recibe una operación REST; la autentifica, autoriza y valida, y tras esto, actualiza la información de cada objeto en el **etcd**. Este flujo se representa en las Figura 14 y Figura 15.

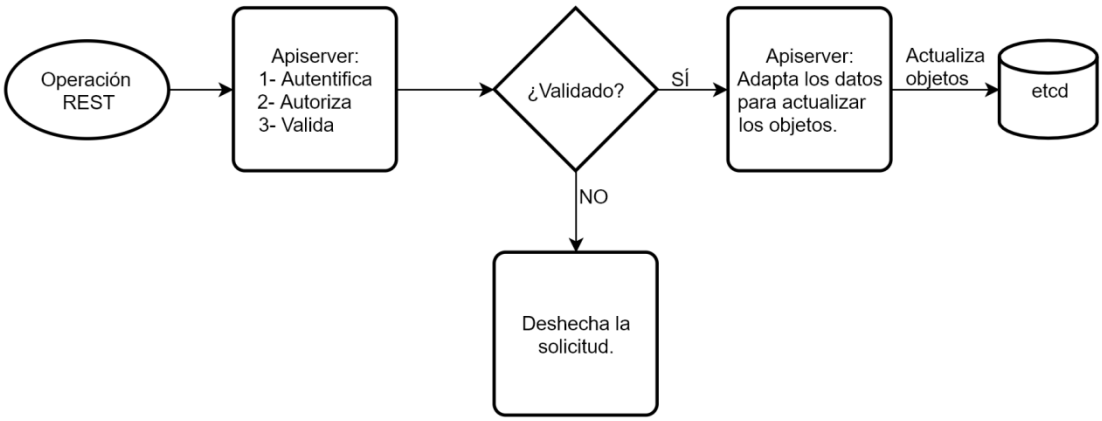


Figura 14 Flujo de acción del apiserver

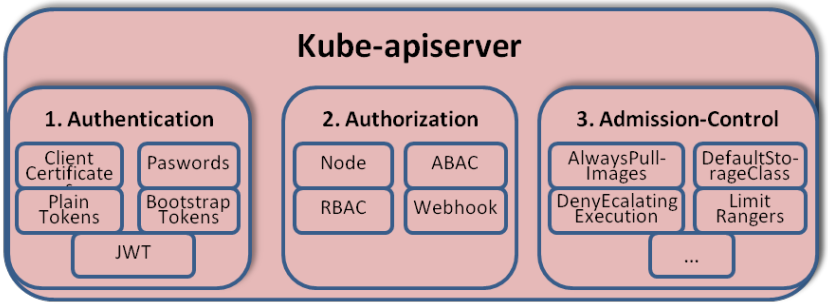


Figura 15 Descripción gráfica de varios elementos que forman el apiserver

4.2.1.1.2 Etcd

Es una base de datos del tipo clave/valor (etc) para sistemas distribuidos, de ahí el 'd'. Un etc tiene datos de configuración de un sistema, mientras que el etcd tiene información de todo el clúster y de las aplicaciones que está ejecutando. El etcd de k8s tiene las siguientes características (Kubernetes, 2019c):

- Almacena **metadatos** de forma robusta y tolerante a fallos (fault tolerant).
- Almacena datos (objetos del sistema) en forma de **clave/valor**, donde una **clave** es un ID para Poder R/W datos definidos directa o indirectamente por el usuario.
- Almacena los metadatos en forma de **versiones**. Cada vez que se modifica el valor correspondiente a una clave se incrementa el valor de la versión. Al crear una clave se le asigna el valor inicial 1 y el valor de la versión de las claves eliminadas es 0.
- Al ser **multiversión**, el etcd recuerda el valor de las anteriores versiones. Luego, los valores no se modifican, sino que se genera una nueva versión. Siendo estas versiones accesibles.
- Permite réplicas para **alta disponibilidad**, pero necesita de un balanceador de carga, como se muestra en la Figura 16.

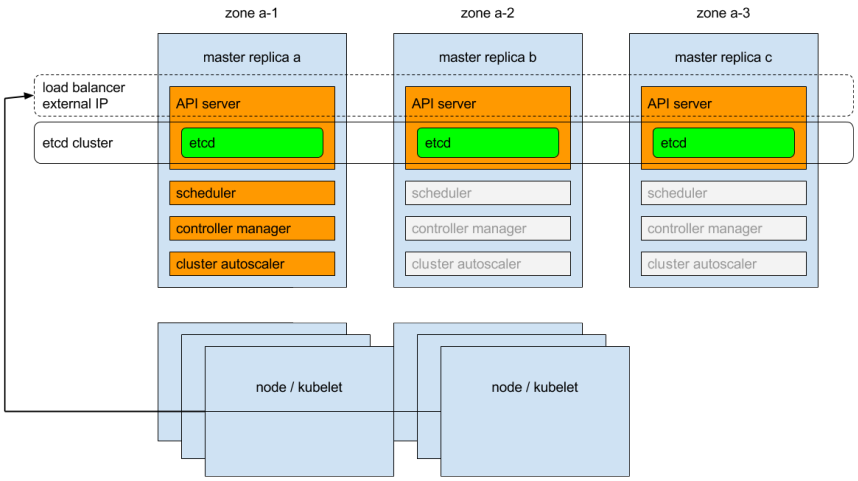


Figura 16 Escalado horizontal del etcd para alta disponibilidad

- Las tareas más habituales con el etcd son; solicitud de servicios, búsqueda de pares clave/valor, escribir datos o vigilar (watch) actualización de datos. Esta última opción se implementa a través de elementos conocidos como shared informers.

K8s guarda en el etcd datos de configuración que luego utiliza para búsqueda de servicios y gestión del clúster (p.e. para planificación), y también el estado del clúster. Gracias al interfaz de vigilancia (**watch API**) de la etcd se puede monitorizar el estado, pudiendo detectar cambios, dando la opción a los controladores de Poder actuar avisándolos mediante eventos que generan los shared informers.

4.2.1.1.3 Kube-controller-manager

El controller manager es el componente que engloba bajo un mismo proceso a los controladores del maestro. En k8s se considera **controlador** al componente que comprueba cíclicamente el estado actual del clúster, y propone acciones que pasen del estado actual al deseado.

El controller manager monitoriza los recursos en los que está interesado definiendo un informador compartido (**shared informer**) para cada recurso. Para gestionar estos informadores, se dispone de manejadores de eventos (**event handler**). Cada informador contiene:

- Reflector: se encarga de hacer un List&Watch de los datos del recurso en el que está. Gestiona todas las instancias de un recurso, con las siguientes acciones:
  - List: Obtiene una lista del estado actualizado de los Pods.
  - Watch: Supervisa si se han dado cambios en el estado de alguno de los Pods.
  - Push: actualiza esta información en un almacenamiento del informador (este elemento se explica en el siguiente punto).
- Store: gracias al reflector, el estado de los Pods está actualizado. Esta parte de la memoria solo se comparte con el resto de informadores del clúster. Cada vez que se da un cambio de estado, se avisa al gestor de evento correspondiente.
- Cada gestor de evento avisa a su controlador, que gestiona estos eventos con una cola de tareas (del tipo FIFO, con bloqueos).
- Por último, los controladores analizarán el evento recibido y propondrán una acción para que el estado deseado y actual concuerden.

Estos controladores se disparan por nivel (**level-triggered**). Es decir, comprueban un rango de valores, no un valor límite (edge-triggered)**.** El apiserver implementa funcionalidades del ciclo de vida (lifecycle) y de la lógica del API de K8s por medio de los siguientes componentes, que como se ha dicho, se ejecutan como un único proceso.

En la Figura 17 se muestran los componentes del Kube-controller-manager.

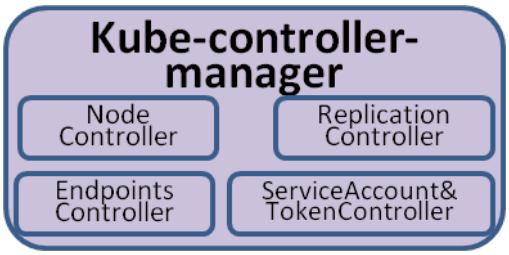


Figura 17 Componentes del Kube-controller-manager

4.2.1.1.3.1 Node Controller

Es el componente del maestro que se encarga de controlar varios aspectos de los nodos trabajadores (Kubernetes, 2019c):

1. Cuando un nodo se registra en el sistema este controlador le asigna un **bloque de CIDR (**Classless Inter-Domain Routing).
2. **Monitoriza el estado de los nodos.** Lo hace por defecto cada 5s (se puede configurar con --node-monitor-period). Si tras 40s, no se recibe un heartbeat del nodo, cambia el estado del nodo (NodeStatus) de NodeReady a ConditionUnknown. Por último, si tras 5 min no se ha recuperado el nodo, desaloja los PODs que se ejecutaban en él. A partir de la versión 1.13 el heartbeat del nodo es un namespace llamado kube-node-lease, que está en un objeto asociado al nodo. El nodo actualiza este namespace periódicamente. Se utiliza este método para aligerar la sobrecarga, dando más rapidez (interesante en la adaptabilidad).
3. **Actualiza la lista de nodos** disponibles. Si alguno deja de estarlo, se encarga de eliminarlo de la lista.

4.2.1.1.3.2 Replication Controller/ ReplicaSet

Se encarga de asegurar que el número de réplicas de cada POD configurado en registro, coincide y se ejecutan correctamente. Esta gestión la hace para que la ejecución global del sistema se mantenga estable. Si una réplica cae, se elimina o se termina, manda iniciar otra nueva en su lugar (Kubernetes, 2019).

El controlador de replicación (**replication controller**) no se encarga de comprobar si las réplicas siguen vivas, sino que supervisa su estado. Está diseñado como un componente al que se le pueden ir agregando funcionalidades de más alto nivel, como auto escalado de réplicas en función de diferentes criterios introducidos por el usuario, nuevos algoritmos de planificación…

K8s propone implementar la funcionalidad de estos controladores a través de despliegues, donde existe otro controlador que gestiona tanto lo relativo al despliegue como a todas las funcionalidades del controlador de replicación, el llamado **ReplicaSet**. No obstante, se menciona este controlador por formar parte de los componentes principales de la arquitectura inicial de k8s.

4.2.1.1.3.3 Endpoints Controller

Es un elemento que K8s crea automáticamente cada vez que crea un servicio. Se encarga de controlar si se dan cambios en los **Endpoint** periódicamente. Siendo un Endpoint una colección de Pods que implementan la funcionalidad de un servicio descrito en el etcd. (Kubernetes, 2019).

4.2.1.1.3.4 Service Account & Token Controllers:

El **controlador token** y las **cuentas de servicios** permiten que un proceso de usuario o una aplicación que se ejecuta en un Pod puedan usar el API del apiserver ofreciéndoles una identidad como parte del clúster. Es un mecanismo necesario de comunicación con el apiserver (ver Figura 12). Dependiendo de la fuente desde la que se esté dando la comunicación con el apiserver, este lo autentificará de una forma:

- Como **usuario**, a través del kubectl, el apiserver autentifica la identidad del usuario por defecto con Poderes de administrador (esto se puede configurar).
- **Desde procesos** de los contenedores **(con UIs o librerías)**, los cuales se pueden comunicar directamente con el API. Pero para Poder identificar estos procesos como parte del sistema k8s, necesitan una cuenta de servicio. Con esto el apiserver los autentifica y permite el acceso. Cuando se crea un Pod, se le asigna una cuenta de servicio por defecto llamado default.

Las cuentas de servicio tienen nombres únicos, que se dan a cada Pod en su creación. El elemento encargado de esto es el **controlador de admisión** (**Admission controller),** que está en el apiserver. La Figura 18presenta de forma gráfica las funciones de las cuentas de servicios.

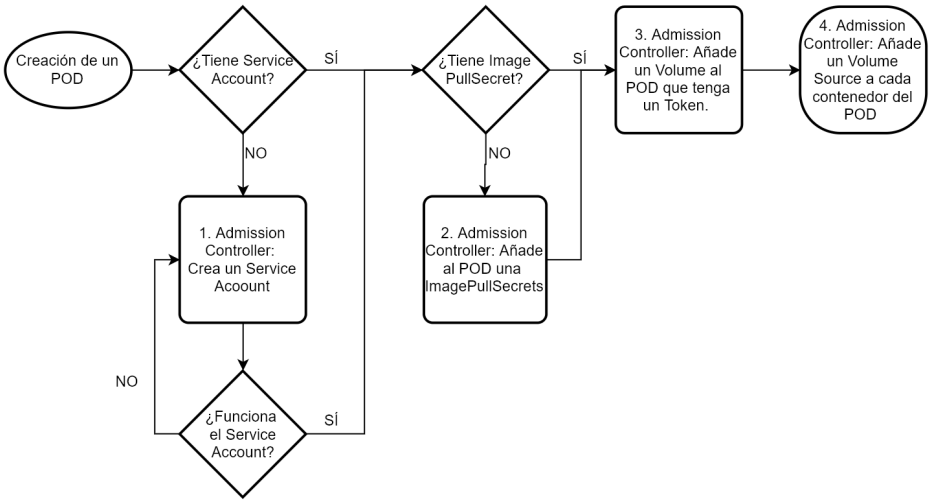


Figura 18 Funciones del Admision Controller

La gestión de los service account recae sobre **Token controller** y **service account controller**.

4.2.1.1.3.4.1 Token Controller

Es una parte del gestor de controladores, controller manager, pero a diferencia de otros controladores, es asíncrono. Crea un secreto (Secret[1]) para cada service account creado (Kubernetes, 2019c). Con esto se consigue autentificar el acceso al API. En caso de que se haya eliminado un Pod, este elimina el secreto asociado a la cuenta de servicio de ese Pod.

4.2.1.1.3.4.2 Service Account Controller

Gestiona los nombres de las cuentas de servicio. Asegura que al menos haya una con el nombre por defecto para cada namespace.

4.2.1.1.4 Cloud-controller-manager

Este elemento es una evolución de la arquitectura inicial de k8s. Surge para distribuir las funciones del kube controller manager y se encarga de ejecutar bucles de control en cargas de trabajo de la nube. Esta distribución deja a los proveedores de servicios cloud evolucionar a un ritmo diferente al que lo hace el core de k8s.

El Cloud-controller-manager realiza funcionalidades análogas a las del kube-controller-manager, ya que implementa sus mismas librerías, pero centrado en la nube. Además, permite trabajar con diferentes proveedores cloud.

[1] *Un Secreto es un objeto que contiene una pequeña cantidad de datos confidenciales, como una contraseña, un token o una clave ssh. Dicha información Podría introducirse en una especificación de Pod (spec) o en una imagen. Pero ponerlo en un objeto Secreto permite un mayor control sobre cómo se usa y reduce el riesgo de exposición accidental.*

4.2.1.1.5 Kube-scheduler

La función principal de este componente es la de comprobar si se han creado nuevos Pods, aún sin nodo asignado, y asociarlos al nodo más apropiado.

La búsqueda del nodo más apropiado recae sobre diferentes criterios, configurables por el usuario. Este debe tener en cuenta los requisitos de recursos individuales y colectivos, los requisitos de calidad de servicio, las restricciones de hardware / software / políticas, las especificaciones de afinidad y anti-afinidad con otros POD, la localidad de datos, la interferencia entre cargas de trabajo, los plazos, etc.

4.2.1.1.5.1 Cómo asocia PODs a nodos (binding):

El scheduler está continuamente monitorizando, gracias a un informador compartido (shared informer), el registro de nuevos Pods en la etcd a los que aún no se les ha asociado nodo en el que ejecutarse. Se encarga de buscar entre todos los nodos existentes en el sistema los que se consideran candidatos para alojar la ejecución del Pod. Es capaz de conocer los nodos candidatos mediante llamadas al apiserver, consultando las características de cada nodo uno a uno. Para la búsqueda nodo a nodo ejecuta un algoritmo de turno rotatorio (Round Robin). Este funcionamiento centralizado, puede llevar a un funcionamiento no deseado en caso de sobrecarga y es donde este trabajo se centra principalmente. De entre todos los nodos candidatos, selecciona el más adecuado en función del criterio establecido en la configuración del Pod. Por último, crea una asociación Pod-nodo conocida en el sistema como **binding** (Tua, 2019). La ilustración 19 representa la secuencia de acciones para asignar el Pod a un nodo.

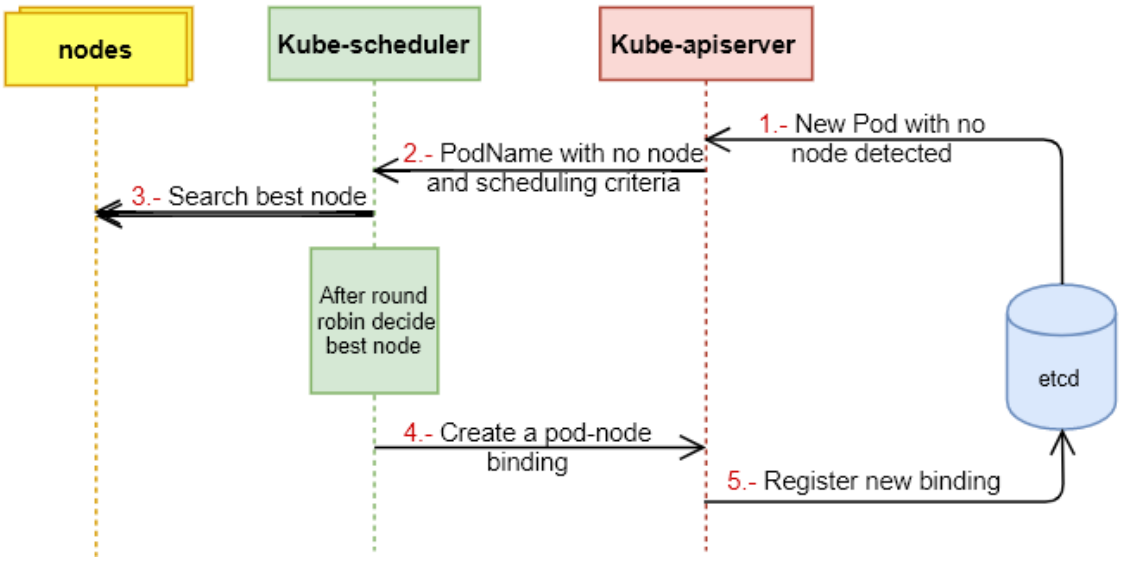


Figura 19 Funcionamiento general kube-scheduler

4.2.1.1.5.2 Tipos de restricciones de ejecución de un Pod a nodo.

K8s ofrece la posibilidad de configurar los Pods para que se ejecuten en un nodo o nodos en concreto. Hay diferentes formas de restringir la ejecución, pero todas se hacen desde las llamadas etiquetas (**labels**[2]**)**. Se puede caracterizar a un Pod con restricciones de ejecución de diferentes formas, como (Kubernetes, 2019a):

- **nodeSelector**: es la forma de restricción más simple y recomendada por k8s. Forma parte de la configuración de un Pod (spec), y se indica en forma de clave/valor. Por ejemplo, la Figura 20 representa la forma de indicar que un Pod necesita ejecutarse en un nodo que tenga un disco ssd:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

Figura 20 Configuracion de un Pod con nodeSelect

- **Node restriction**: es una etiqueta que permite dirigir Pods a nodos específicos o grupos de nodos.
- **Affinity/anti-affinity**: proporciona una forma muy sencilla de restringir los Pods a nodos con etiquetas concretas. Consta de dos tipos de afinidad:
  - **node affinity**: es como el nodeSelector arriba expresado pero más flexible, porque no tiene que cumplir completamente la restricción. Este concepto tiene dos etiquetas:
    - **requiredDuringSchedulingIgnoredDuringExecution**: esta es la etiqueta que más se acerca a la de selección de nodo (nodeSelector), a diferencia de que tiene una sintaxis más expresiva, ya que se requiere durante la acción de despliegue.
    - **preferredDuringSchedulingIgnoredDuringExecution**: este sería más flexible, señala preferencia sin llegar a restricción.
  - **inter-Pod affinity/anti-affinity**: este concepto trabaja con reglas del tipo; el Pod XX debería/no debería ejecutarse en nodos en los que estén ejecutándose Pods que cumplan con los labels YY.

[2] ***labels**** **Las etiquetas** *son pares clave / valor que se caracterizan objetos, como Pods. Las etiquetas se utilizan para especificar atributos de identificación de objetos que son significativos y relevantes para los usuarios, pero que no implican directamente semántica para el sistema central.*

Un concepto con el que también se puede trabajar para restringir son los **taints/tolerations** (Kubernetes, 2019). Esto es lo opuesto al anterior punto, permite a los nodos rechazar ciertos Pods.

- **Taints**: capacidad de los nodos de rechazar Pods. Se aplica en los nodos.
- **Tolerations**: permite a los Pods ser planificados en los nodos que cumplan las condiciones. Se aplica a los Pods.

Taints/tolerations trabajan juntos para asegurar que no se planifican Pods en nodos que inapropiados.

4.2.1.2 Componentes de nodo

Un nodo del clúster K8s tiene los servicios necesarios para ejecutar contenedores de aplicaciones y ser administrado desde el maestro. El tiempo de ejecución de un nodo es la suma del tiempo de ejecución de los contenedores que aloja, el kubelet y kube-proxy (Kubernetes, 2019e).

A diferencia de los Pods y los servicios, k8s no crea nodos, sino que se crean externamente por proveedores de la nube como **GCE** (Google Compute Engine) o existen en el clúster de máquinas físicas o virtuales. Por tanto, es el nodo el que se une al clúster y como resultado se crea en el sistema un objeto que lo representa. Tras la creación, el plano de control de k8s comprueba si el nodo es válido o no. Para comprobar su validez, analiza si se están ejecutando en él ciertos servicios. De ser válido, se convierte en un nodo apto para alojar Pods.

4.2.1.2.1 Kubelet

Es el controlador principal en todos los nodos trabajadores del clúster. Su función es la de leer periódicamente los **PodSpecs** de cada Pod y asegurar que los contenedores que hay en cada Pod se ejecutan correctamente. Además, se le puede enlazar un **cAdvisor**, con el que es capaz de monitorizar la actividad del nodo y obtener métricas. El cAdvisor es un componente que obtiene toda esa información y la procesa del componente linux **cGroups** (Zhang, 2017).

Así como Docker es la herramienta encargada de la ejecución de los contenedores, Kubelet tiene un papel análogo en la gestión de los Pods de K8s. Donde para Docker la unidad de gestión es el contenedor, para K8s es el Pod.

Entre la variedad de funciones que tiene Kubelet, se han seleccionado algunas representativas para dar una perspectiva general del rango de actuación de este elemento sobre los PODs. Se representa el funcionamiento general de este componente en la Figura 22.

Kubelet es un elemento que interviene en el despliegue de cualquier aplicación, y aunque el admission controller (componente del apiserver, ver Figura 15) puede rechazar la ejecución de Pods en ciertos nodos, es este elemento en última instancia, quien decide si un Pod se ejecuta o no en el nodo.

También es el encargado de actualizar el estado de los Pods desplegados en su nodo, que está compuesto por el estado de sus contenedores. Para realizar un diagnóstico que determine el estado, el kubelet llama a un controlador implementado por cada contenedor. Existen tres tipos de controladores:

- **ExecAction**: ejecuta un comando en cada contenedor. El comando comprueba que el contenedor tiene una ejecución adecuada y devuelve un 0.
- **TCPSocketAction**: comprueba el puerto del contenedor realizando un TCP check contra la IP del mismo, si el puerto está abierto todo está correcto.
- **HttpGetAction**: comprueba el estado mediante una solicitud http tipo get contra la IP del contenedor en un puerto específico. Tiene un estado adecuado cuando el código de respuesta está entre 200 y 400. Esta solicitud se ilustra en la Figura 21.

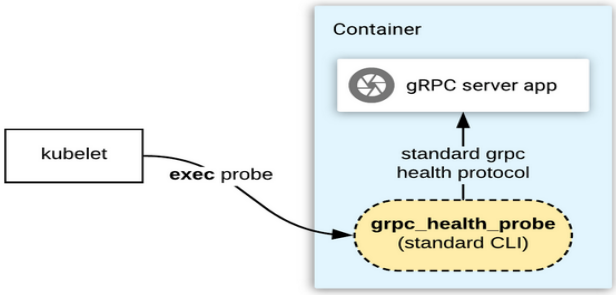


Figura 21 Health check (Balkan, 2018)

El Kubelet realiza dos tipos de pruebas para obtener el estado de cada contenedor, cuye resultado puede ser éxito (success) (el contenedor pasa el diagnostico), fallo (failure) (no lo pasa), desconocido (unknown) (ha fallado el diagnostico). Por medio de los tres controladores anteriores, el kubelet puede realizar dos tipos de pruebas:

- **livenessProbe**: Indica si el contenedor se está ejecutando.
- **readinessProbe**: comprueba si el contenedor está listo para ejecutar la funcionalidad que requieren las aplicaciones que aloja.

Además de lo presentado, otra función de Kubelet es la de recolector de basura (**Garbage collection**) (Kubernetes, 2018). Es una herramienta que permite limpiar del sistema contenedores e imágenes de contenedores. Limpiar significa a eliminar contenedores e imágenes de contenedores que no se usen. Con esto se reduce la carga de memoria del sistema.

Por último se destaca también la función de gestor de recursos (**Resource Handling)** del Kubelet (Kubernetes, 2019d). Como los recursos de un nodo son variables en el tiempo, Kubelet tiene que ser capaz de asegurar la estabilidad del nodo al que representa. Para ello, es capaz de aplicar diferentes políticas de desalojo de Pods.

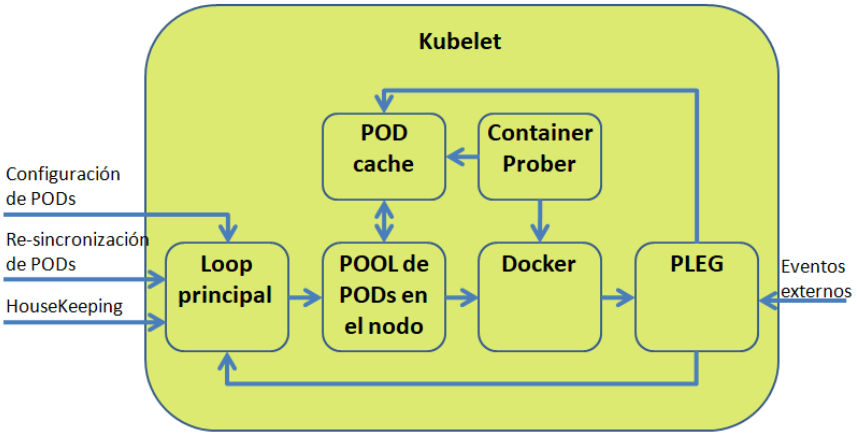


Figura 22 Representación gráfica del funcionamiento de Kubelet

Por último, cabe indicar que aunque se sitúe a Kubelet dentro del sistema de K8s, en realidad es un controlador de contenedores. Se Podría llegar a utilizar de manera aislada para gestionar contenedores ejecutándose en un único nodo, fuera del sistema de k8s (Marhubi, 2015).

4.2.1.2.2 Kube-proxy

En K8s puede haber diferentes tipos de proxys, pero el Kube-proxy es un componente que se ejecuta como Pod (en realidad es un proceso Daemon) que tienen todos los nodos. El concepto de servicio presentado anteriormente, requiere este elemento para mantener reglas de conexiones entre Pods.

El Kube-proxy muestra los servicios tal y como se indican en el API, y establece conexiones con los Pods que los quieren utilizar. Se pueden realizar diferentes configuraciones del proxy a través del apiserver, p.e. para seleccionar las estrategias de conexión (Pod-servicios). Trabaja con diferentes protocolos; UDP, TCP y SCTP. No trabaja con http.

El proxy realiza las conexiones a las ip virtuales de los servicios, con una IP rules y las redirige a los **backedns** que componen cada servicio.

4.2.1.2.3 Container Runtime (Docker)

El runtime del contenedor es el software que se encarga de ejecutar los contenedores. Kubernetes admite varios en tiempo de ejecución: Docker, containered, cri-o, rktlet y cualquier implementación del CRI (Interfaz del runtime del contenedor) de k8s. En este proyecto se trabaja con Docker.

4.2.1.2.4 Addons (Complementos)

Los complementos son Pods y servicios que implementan nuevas características en el clúster. Existen entre otros: DNS, web UI (Dashboard), flannel…

4.2.2 Comunicaciones Maestro-Nodo

Este apartado presenta en el primer subapartado las comunicaciones que se pueden dar desde los nodos trabajadores al maestro, y en el sgundo las que se dan desde el maestro a los nodos trabajadores.

4.2.2.1 Nodos trabajadores a Maestro.

Todas las rutas de comunicación desde nodos trabajadores del clúster, al maestro terminan en el apiserver (ninguno de los otros componentes del maestro está diseñado para ofrecer servicios remotos). Los nodos del clúster deben contar con un certificado público y credenciales válidas para Poder conectarse de forma segura al apiserver (Kubernetes, 2019g).

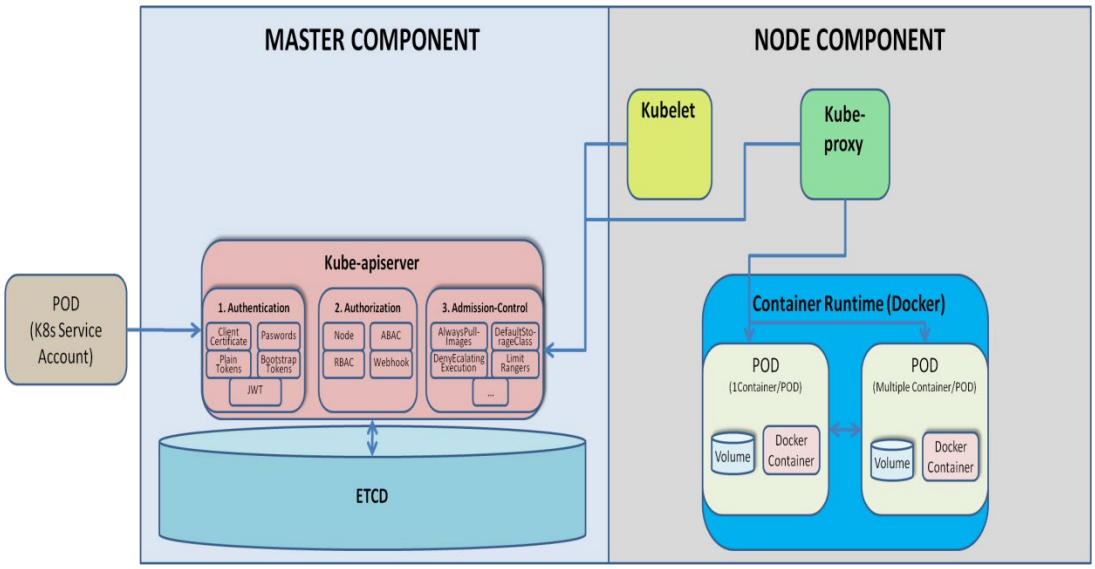


Figura 23 Comunicaciones del nodo trabajador al maestro

Los Pods que desean comunicarse con el Apiserver pueden hacerlo a través de una cuenta de servicio (service account). Anteriormente se presentó un servicio como, una abstracción que define un conjunto lógico de Pods y una política mediante la cual acceder a ellos. Pero para Poder comunicarse con un servicio, éste se configura con una dirección IP virtual que redirige la conexión (a través de kube-proxy) al punto final de HTTPS en el apiserver.

Como resultado, el modo operativo del sistema para las conexiones desde el clúster (elementos del nodo y Pods que se ejecutan en los nodos) al maestro está protegido, permitiendo así realizarse desde redes no confiables y / o públicas.

4.2.2.2 Maestro a nodos trabajadores.

Hay dos rutas de comunicación principales: del apiserver al Kubelet y del apiserver a nodos, Pods y servicios, a través del kube-proxy.

4.2.2.2.1 Apiserver a Kubelet

Las conexiones del apiserver al kubelet se utilizan para:

- Detectar registros de Pods.
- Edición de la configuración (a través de kubectl) de Pods en ejecución.
- Proporcionar a Kubelet una funcionalidad de redireccionamiento de puertos.

Estas conexiones terminan en el punto final HTTPS del kubelet. Por defecto, el apiserver no verifica el certificado de servicio de kubelet, quedando esta información vulnerable ante amenazas (p.e. en conexiones desde redes no confiables y / o públicas). No obstante, k8s ofrece mecanismos para asegurar este defecto, como el uso de un túnel SSH entre el apiserver y el kubelet.

4.2.2.2.2 Apiserver a nodos, Pods y servicios.

Las conexiones del apiserver a un nodo, Pod o servicio son conexiones HTTP simples y, por lo tanto, no están autentificadas ni encriptadas.

Se pueden realizar conexiones seguras mediante llamadas https, que aunque cifran las comunicaciones, no realizan servicios de autentificación de credenciales. Luego, no ofrecen garantías de conexión segura, y esto es un defecto actual de k8s. Sin embargo, k8s admite túneles SSH para proteger las rutas de comunicación del Maestro -> Clúster. En esta configuración, el apiserver inicia un túnel SSH a cada nodo del clúster y pasa todo el tráfico destinado a un kubelet, nodo, Pod o servicio a través del túnel. Este túnel garantiza que el tráfico no se exponga fuera de la red en la que se ejecutan los nodos.

4.2.3 El modelo de red de k8s: Redes del cluster

K8s no ofrece implementaciones de red, simplemente define un modelo de red con ciertos requisitos y permite que otras herramientas lo implementen (Kubernetes, 2019b). En este proyecto se utiliza Flannel(Chopra, 2019) para implementar el modelo de red de k8s. Se elige esta opción por ser una opción simple y con bastante madurez en el uso de k8s. Además ofrece un buen funcionamiento sobre el sistema operativo linux, utilizado en este trabajo.

Flannel es una red superpuesta (**overlay network**[3] (Hesselbach Serra & Altés Bosch, 2002)**)**, es decir, es un conectable (plugin) para las comunicaciones que cumple con los requisitos de red de k8s:

- Todos los contenedores **(contenedor-contenedor)** se tienen que Poder comunicar entre ellos sin necesitar la traducción de direcciones de red (*Network Address Translation* **NAT)**.
- Comunicación **nodo-contenedores** sin necesidad de NAT.
- La IP de un contenedor es la que ven los demás contenedores.

Flannel propone crear un nivel de red que se ejecuta sobre la red del nodo (esta es la definición de **overlay network**). Para esto, asigna una IP a todos los PODs en esta nueva red, permitiendo su comunicación. Esta herramienta abstrae al usuario de como se realiza dicha comunicación. Se explica la implementación de red propuesta por Flannel con un ejemplo representado en la Figura 24.

Se supone un único POD por nodo, por darle simplicidad a la imagen y a la explicación. Si hubiese un segundo POD en alguno de los nodos por ejemplo, habría otro docker0 con la dirección 100.96.2.1/24.

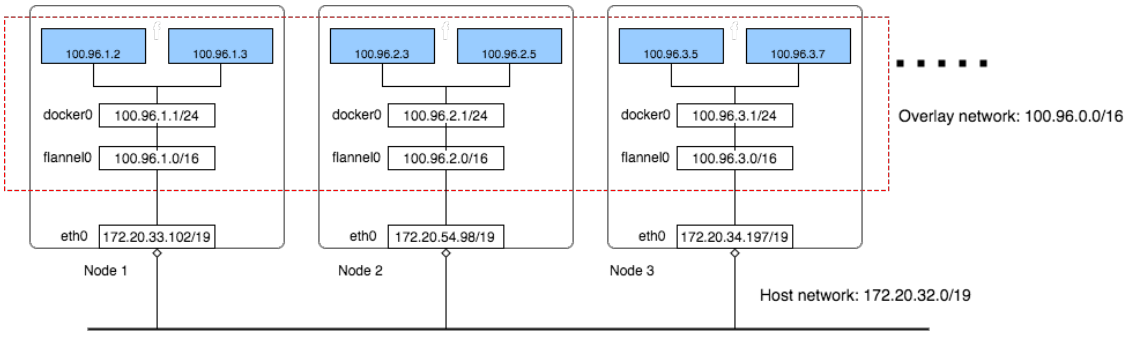


Figura 24 Implementación del modelo de red de k8s (Flannel)

[3] **Overlay network*:** Una **red superpuesta** (en inglés, overlay network) es una **red** virtual de nodos enlazados lógicamente, que está construida sobre una o más **redes** subyacentes (underlying network). Se dice que los nodos de la **red superpuesta** están conectados por enlaces virtuales.

Se identifican tres redes en este clúster:

1. **Red del Host**: (172.20.32.0/19): los host (diferentes nodos) se pueden comunicar entre ellos porque están en la misma **LAN**.
2. **Red** superpuesta **de Flannel**: Flannel se encarga de crear una nueva red, en este caso en 100.96.0.0/16, puede tener 2^{16} direcciones. Esta red se extiende por todos los nodos de K8s. Se encarga de dar una IP diferente a cada nodo.
3. **Red de docker** en cada nodo (HOST): en esta red Flannel asigna una dirección a cada POD en el nodo (host). En el rango de 100.96.x.0/24, puede dar 2^{8} direcciones. El bridge docker0 utilizará esta red para crear nuevos contenedores. Con esta estrategia se consigue que cada contenedor del clúster tenga una IP diferente. Pero todas estas direcciones recaen sobre la red superpuesta 100.96.0.0/16.

Con el uso de Flannel se habilita:

1. Comunicación **contenedor-contenedor** dentro del **mismo nodo**: gracias al bridge docker0.
2. Comunicación **contenedor-contenedor** en **distinto nodo**: Fannel utiliza la kernel route table de raspbian (es una tabla donde se indica el camino que se debe seguir para llegar a la dirección IP del contenedor que se solicita) y encapsula el mensaje en formato UDP. Raspbian es un sistema operativo basado en Debian y por tanto en linux empleado en este proyecto. La Figura 25representa este tipo de comunicación.

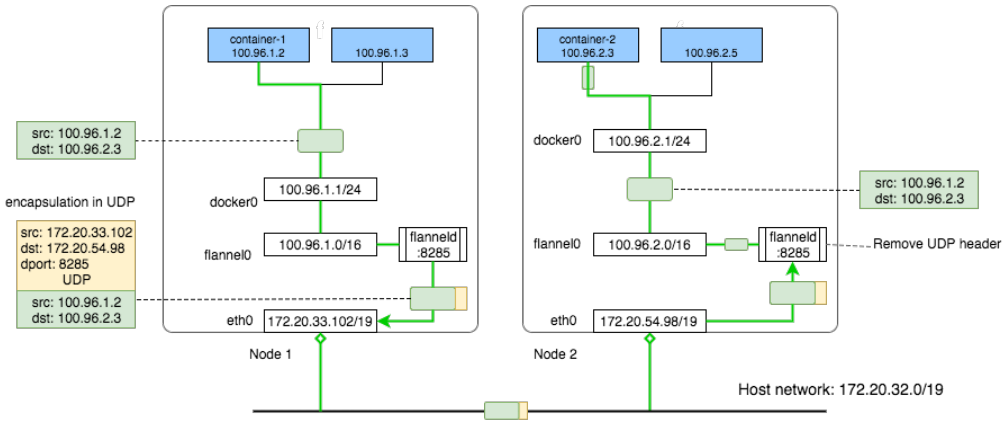


Figura 25 Comunicación contenedor-contenedor en distintos nodos

4.3 Análisis de opciones de extensión/modificación de la arquitectura orientado al despliegue.

Una vez presentados los componentes de la plataforma k8s, conviene conocer cuáles de ellos intervienen en un proceso de despliegue. El despliegue de una aplicación, controlador o cualquier elemento susceptible de requerir un nodo en el que ejecutarse, sigue el mismo proceso.

Existen diferentes formas de definir un despliegue. Entre ellas, la opción más común es lanzar la ejecución de un fichero "*.yaml*" desde la línea de comandos kubectl. A continuación, se describe el proceso de despliegue a partir de la definición del mencionado fichero ".yaml", cuyo formato se ilustra en la Figura 26.

```yaml
apiVersion: v1
kind: Deployment
metadata:
  name: v1Deployment
spec:
  replicas: 1
  template:
    spec:
      containers:
      - name: containerName
        image: containerImage
        ports:
        - containerPort: 80
```

Figura 26 Fichero de despliegue en formato .yaml

En primer lugar se define la versión del API k8s que se va a usar (apiversion: v1). A continuación se indica la zona del componente etcd donde se va a registrar la información (kind). En este caso, la que corresponde a despliegues (deployment). El campo metadata identifica el despliegue mediante un nombre (name). El campo spec caracteriza el objeto que se debe desplegar. En este caso un contenedor con una única réplica (replicas: 1). K8s recomienda seguir la filosofía de un contenedor por Pod. No obstante, como ya se ha mencionado, esto es algo que también se puede configurar. El campo containers define el nombre del contenedor y especifica la imagen del contenedor que se debe cargar. Finalmente, la sección ports, permite definir el puerto por el que el contenedor se comunicará (80).

El comando que lanza el despliegue es el siguiente:

kubectl apply -f pathTodeploymentfile.yaml

La ejecución del comando provoca el flujo que caracteriza un despliegue en k8s, mostrado en las Figura 27Figura 28 mediante un diagrama de secuencia UML y que consta de los siguientes pasos.

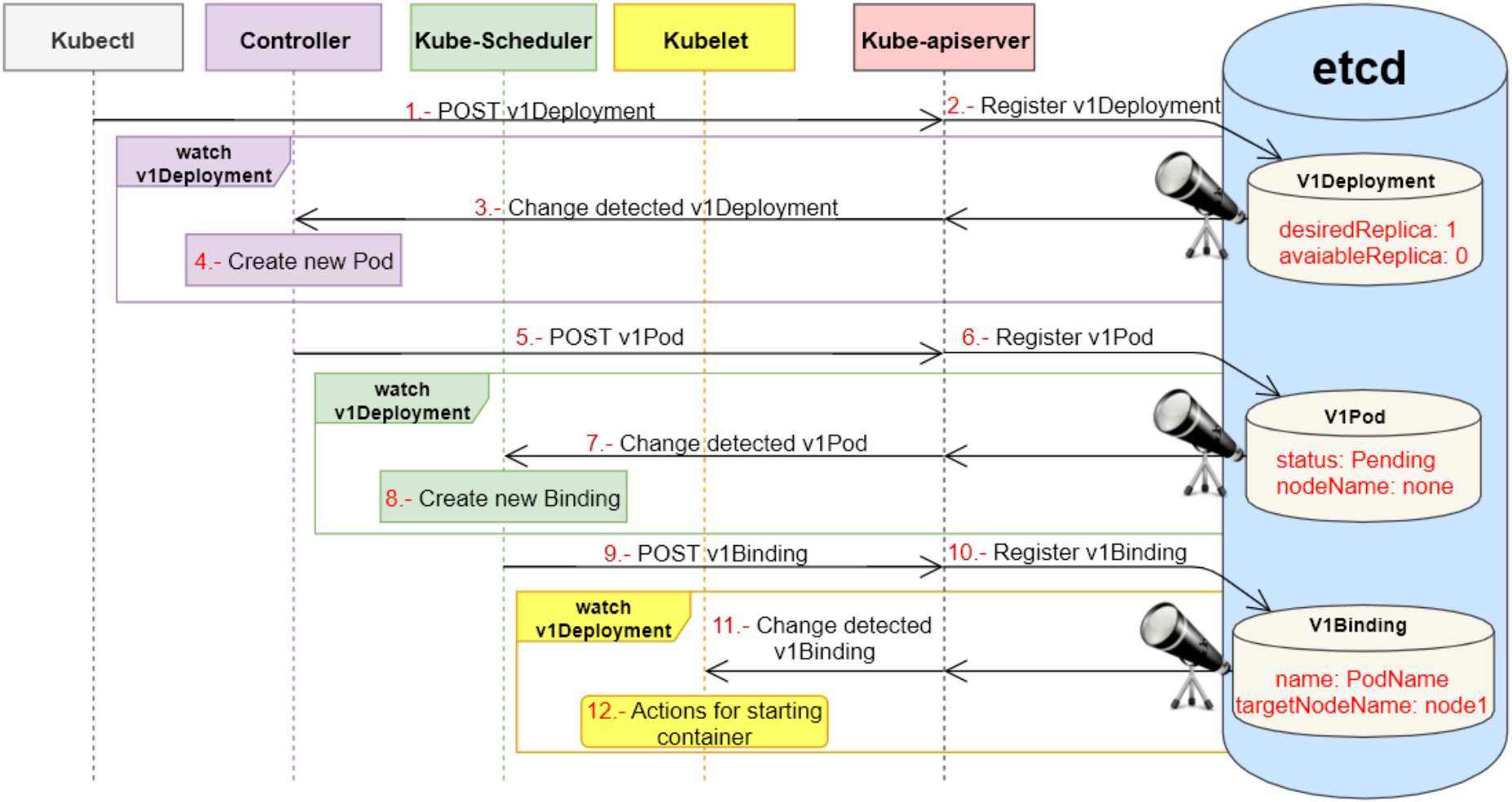


Figura 27 Interacciones entre elementos en un despliegue (PARTE I)

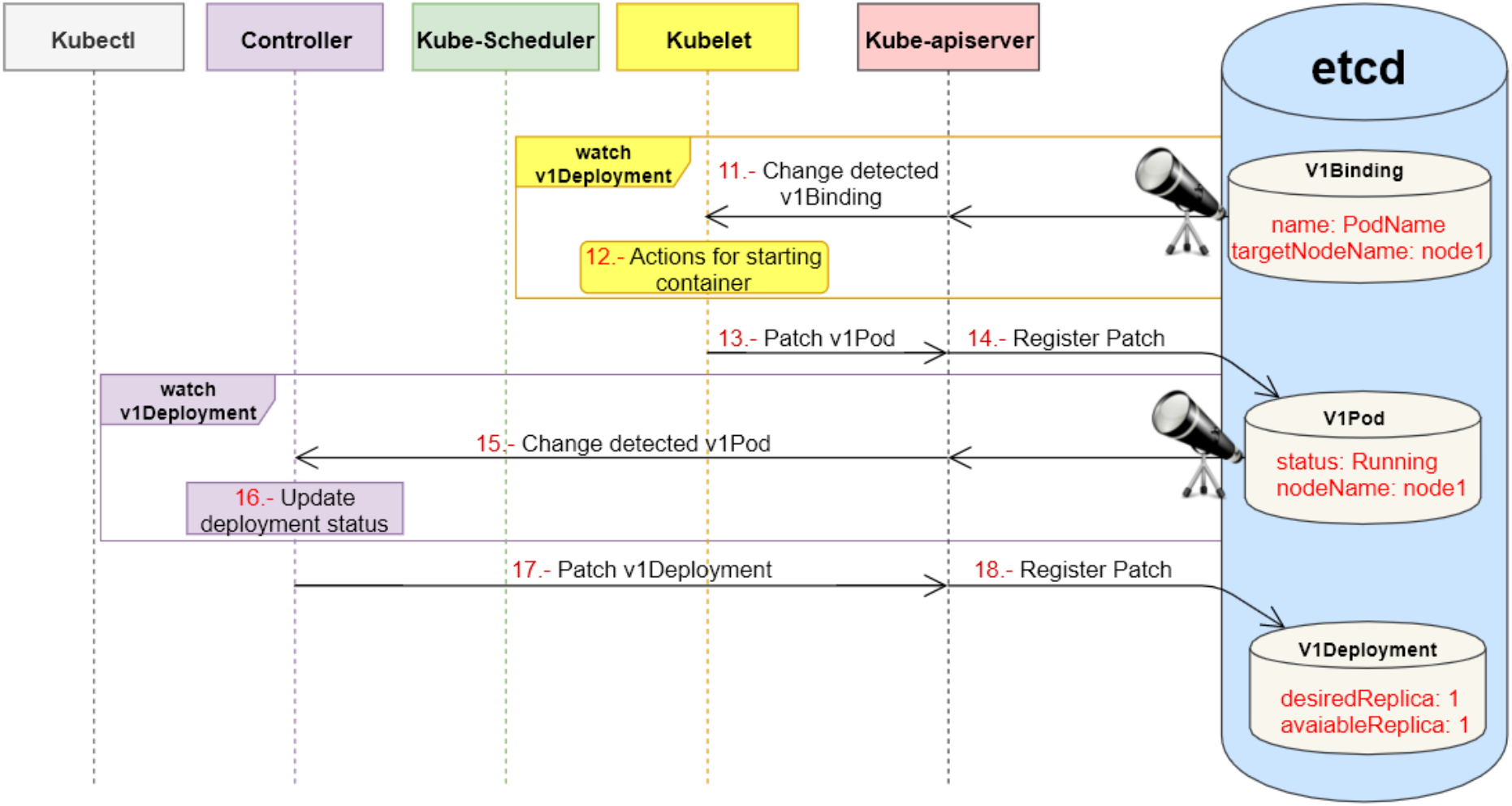


Figura 28 Interacciones entre elementos en un despliegue (PARTE II)

1. Kubectl realiza una llamada tipo REST al apiserver con la información del fichero "*.yaml*" (Paul Morie, 2017).
2. El Apiserver registra la información recibida en formato 'JSON' en el Etcd, en forma de clave/valor como se muestra en la Tabla 4.

Tabla 4 Información de un despliegue en el etcd

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAIABLE	AGE
V1deployment	1	0	0	0	1s

3. Cualquier cambio introducido en el componente etcd puede ser detectado por los controladores a través del recurso monitor. k8s permite establecer filtros de detección/vigilancia. De esta forma el controlador detecta la escritura de nuevas solicitudes de despliegue.
4. En función de la petición detectada el controlador toma una acción. En este caso, crear un Pod en el que se va a alojar el contenedor que se quiere desplegar.
5. La acción de crear un Pod se realiza mediante una llamada REST al apiserver, solicitando la creación de un Pod.
6. El apiserver registra un nuevo Pod en el sistema, al que todavía no se le ha dado ni nombre, ni UID (Unique IDentificator). El Pod se crea con un estado de 'Pending', lo que significa que el Pod ha sido aceptado por el sistema, pero aún no se ha creado la imagen del contenedor.
7. El Scheduler detecta que hay un Pod en estado de 'Pending' y sin nodo asociado.
8. La siguiente acción es generar lo que se conoce como Binding. Consiste en asocuiar un Pod a un nodo en el que se ejecutará. El scheduler, mediante sucesivas llamadas al apiserver, obtiene una lista de nodos candidatos, de los que elige el mñas adecuado en función de un criterio configurable (en el spec de un Pod).
   Este proceso es mejorable ya que la obtención de nodos candidatos se da mediante llamadas http sucesivas al apiserver, pudiendo resultar en un proceso lento.
9. Una vez que el Pod tiene asociado un nodo, el Scheduler solicita al apiserver que cree un objeto en el etcd con las características del Binding. Estas son, el nombre del Pod y el nodo en el que se debe ejecutar.
10. El apiserver registra la llamada del scheduler en la del componente etcd que corresponde.
11. El Kubelet del nodo que debe alojar el Pod detecta la escritura del Binding.
12. Kubelet inicia las acciones para arrancar el contenedor dentro del Pod indicado en el Binding. Para ello accede a la descripción del Pod en el Etcd y obtiene la información acerca del contenedor que debe arrancar. Esta información minimamente contiene:
    a. Nombre del contenedor.
    b. Path de la imagen del contenedor.
    c. Imagen del contendor.
13. Cuando el kubelet ha iniciado el contenedor, solicita una actualización de la información al apiserver del Pod que ejecuta mediante la operación REST Patch.

14. El apiserver procesa la solicitud cambiando, entre otra información, el estado del Pod de 'Pending' a 'Running' y actualizando la localización del Pod en el sistema (nodo que lo aloja).
15. El controlador detecta que el Pod relacionado con el despliegue que monitoriza, ha sufrido un cambio.
16. El controlador comprueba si todos los contenedores que contiene el despliegue están iniciados (estado de 'Running') y el estado en que se encuentran es el estado deseado del despliegue (el registrado por el usuario, en este caso ejecutar un contenedor).
17. Cuando el controlador ha confirmado que el estado deseado y el actual coinciden, solicita una actualización de la información del despliegue.
18. El apiserver recibe la solicitud del paso anterior y actualiza la información del despliegue en el etcd. Los campos actualizados se muestran en la Tabla 5.

Tabla 5 Estado actualizado de un despliegue en el etcd

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAIABLE	AGE
V1deployment	1	1	1	1	10s

Analizada la forma en la que se da un despliegue en k8s, se observa que es el scheduler el que tras sucesivas llamadas REST obtiene la lista de nodos disponibles y toma de forma centralizada la decisión de despliegue. Este proceso centralizado puede ser un inconveniente cuando el despliegue se realiza como consecuencia de una situación detectada en el entorno (por ejemplo, situación de reconfiguración típica en sistemas distribuidos sensibles al contexto).

Desde que se formó k8s como evolución de la plataforma Borg, se sigue la filosofía de descentralizar ciertos procesos de la orquestación de los contenedores. Este proyecto trata de seguir con esa filosofía, y en este caso se propone descentralizar la decisión de selección del nodo mediante negociación distribuida en un sistema multi-agente. Lo que evitaría la obtención de la lista de nodos disponibles por llamadas sucesivas y la toma de decisión en base al criterio seleccionado.

4.4 Integración k8s-JADE: Diseño de la extensión/modificación de arquitectura propuesta

A continuación se presenta una propuesta de paso de decisión centralizada a distribuida. Representada de forma gráfica en las Figura 29 y **¡Error! No se encuentra el origen de la referencia.** .

k8s ofrece la posibilidad de indicar en la configuración de los Pods el scheduler con el que deben ser planificados. Por lo que, crear un nuevo scheduler capaz de reducir la carga del nodo máster puede ser una buena opción. Con la intención de encontrar la forma en la que reducir dicha carga, se describe el flujo de planificación que sigue kube-scheduler a la hora de planificar:

1. Detectar Pods sin nodo asignado.
2. Obtener una lista de estos Pods, y planificarlos uno a uno.
3. Para cada Pod sin nodo asignado detectado se obtienen los nodos candidatos en función del criterio que tenga éste en registro. Puede ser que haya Pods con restricciones de nodo, afinidades a nodos...
4. Se ordena la lista de candidatos en base al criterio configurado en el fichero de despliegue.
5. Por último, se crea un 'Binding'. Es decir, se asocia el Pod al primer nodo de la lista de candidatos.
6. El Binding es oficial en el sistema cuando el Scheduler realiza una llamada REST (de tipo POST) al Apiserver para que lo registre en el Etcd. El Binding es un objeto de api con la información en formato'JSON'.

En el tercer y cuarto punto del flujo el kube-scheduler necesita hacer una serie de consultas al etcd a través del apiserver, para obtener la lista de nodos candidatos y ordenarla.

El nuevo scheduler que se propone va a delegar a los nodos candidatos la decisión del nodo más adecuado. Para obtener la lista de nodos candidatos, no es necesario hacer consultas a la etcd si los propios nodos conocen sus recursos, ya que una única consulta al gestor del sistema MAS (componente DF de JADE) es suficiente, con una comunicación más eficiente. Se propone una negociación entre nodos iniciada por el scheduler entre los candidatos, quedando éste a la espera de recibir un mensaje del nodo ganador. Dadas las características de los agentes JADE como son; autonomía, habilidades sociales, reactividad, movilidad, veracidad, etc. Se presenta como una buena opción para crear un nuevo Scheduler agentificado (agente JADE), dotando así a k8s de la capacidad de realizar negociaciones distribuidas y dejando abierta la puerta a nuevas funcionalidades distribuidas.

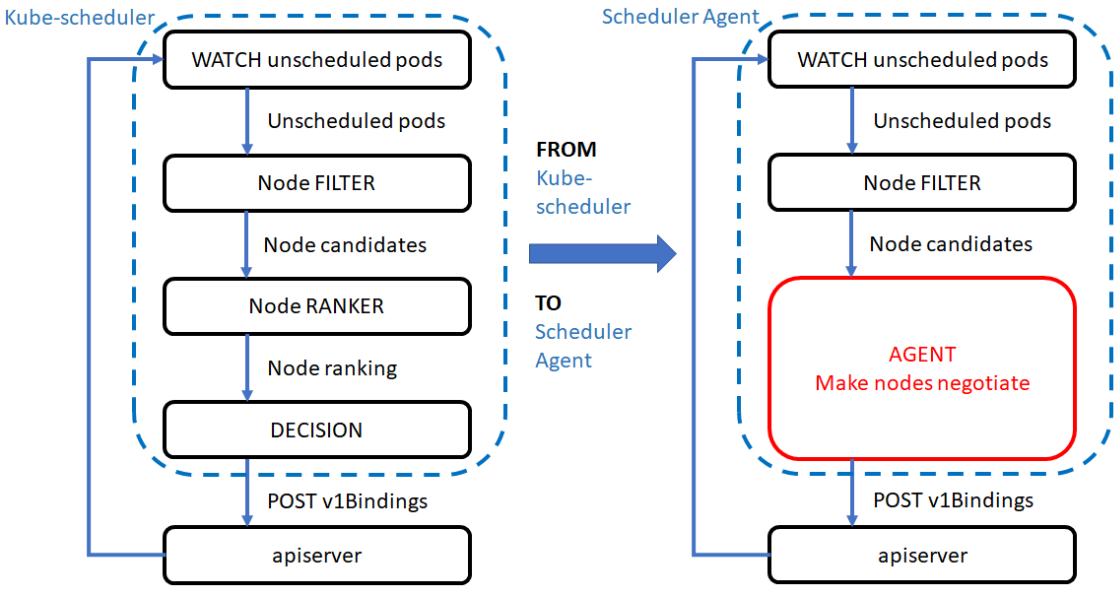


Figura 29 Comparativa kube-scheduler vs scheduler agent.

El kube-scheduler, se ejecuta como un Pod. En consecuencia, el scheduler agentificado también debe hacerlo. A partir de ahora, se referirá al kube-scheduler como scheduler y al scheduler propuesto como scheduler agent.

El scheduler agent debe ser un Pod con la capacidad de planificar. Pero para Poder planificar, tiene que comunicarse con el apiserver. Se recuerda que para Poder comunicarse con el apiserver se necesitan credenciales. Para obtener las credenciales necesarias, el Scheduler Agent se comunicará con el apiserver a través de una librería cliente, siendo la librería la que aporta dichas credenciales. Como se observa en la Figura 13, ésta es una de las opciones que admite k8s.

Se propone una negociación entre nodos iniciada por el scheduler agent. Esto supone que en cada nodo debe haber un agente, que se llamará node agent. Este conjunto de agentes extiende la arquitectura JADE formando un MAS con la estructura que se muestra en la Figura 30).

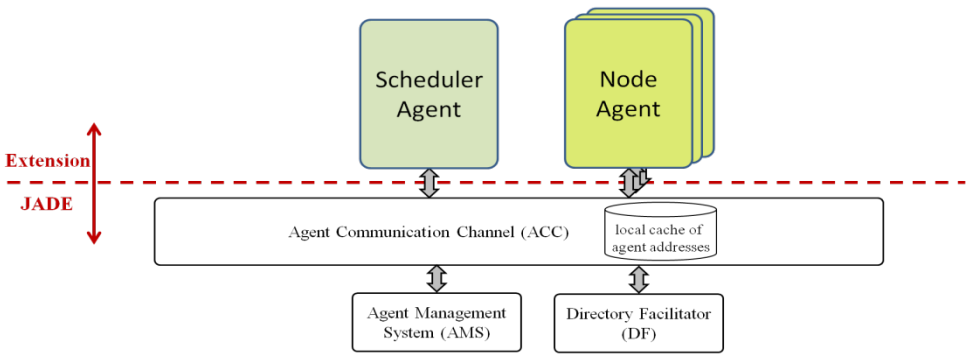


Figura 30 Arquitectura MAS propuesta

4.5 Negociación de agentes JADE

En el sistema MAS presentado, se identifican dos agentes que ejecutan las interacciones necesarias para la planificación en un proceso de despliegue: el scheduler agent y node agent. Estas se representan en la Figura 32. A continuación se presenta el flujo de las interacciones que desarrollan estos agentes, en una negociación de acuerdo con la Figura 31:

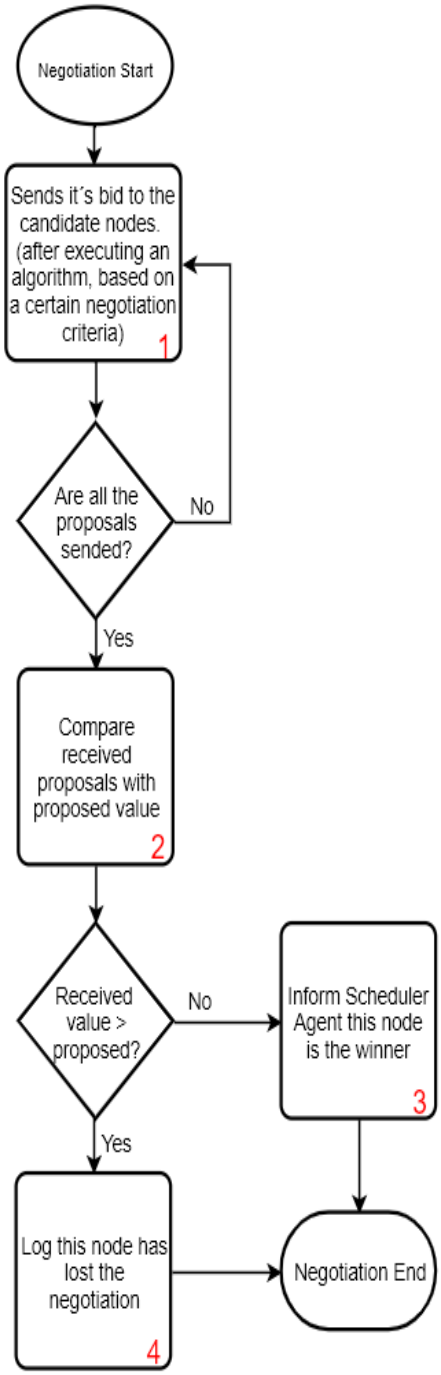


Figura 31 Flujo de una negociación entre agentes JADE

1. Toda negociación es iniciada por el scheduler agent mediante una llamada de propuestas (mensaje ACL con performative 'CFP',Call For Proposal). Este solicita a los nodos que hagan propuestas entre ellos e indica a cada uno con qué nodos deben comunicarse, además del criterio de negociación. El nodo ganador debe ejecutar una acción indicada por el scheduler agent, que en este caso es identificarse como ganador ante este agente.
2. Cuando un nodo recibe el mensaje del scheduler, cambia su comportamiento a uno de negociación, ejecutando un algoritmo con los siguientes pasos:
   2.1. Envío de propuesta: cada node agent envía al resto de nodos candidatos una propuesta con el valor de su memoria libre disponible, mediante un mensaje ACL con performative 'Propose'. Tras solicitar todas las propuestas, se va al siguiente paso.
   2.2. Recepción de propuestas: el nodo analiza una a una las propuestas de los candidatos y si el valor recibido es mayor que el suyo, pierde la negociación, saltando al paso 2.4. El caso de los nodos 2 y 3 en el ejemplo propuesto.
   2.3. Nodo ganador: a este paso solo llegan los nodos que no han abandonado las negociaciones, habiendo recibido las propuestas de todos los candidatos. El nodo informa al scheduler que ha ganado la negociación, en el ejemplo, el nodo 1. Lo hace mediante un mensaje ACL de performative 'Inform'.
   2.4. Fin de la negociación: en este punto se da por finalizada la negociación y el node agent vuelve a su comportamiento por defecto.

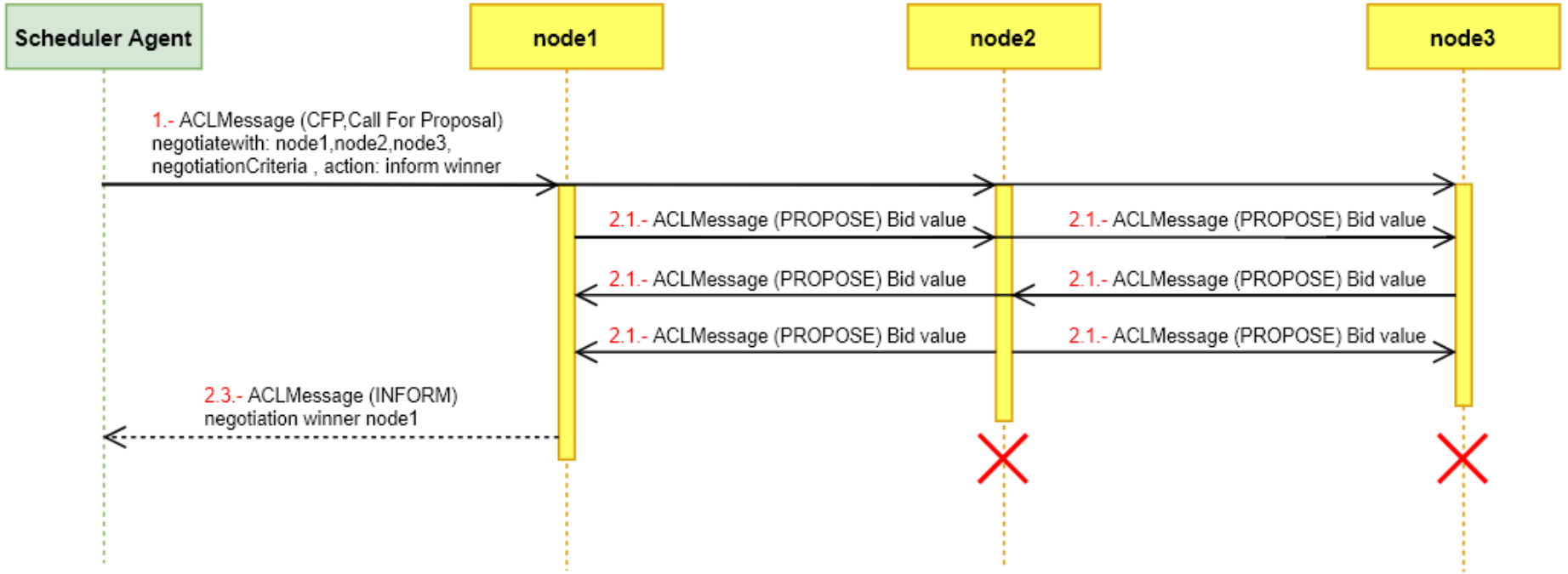


Figura 32 Ejemplo de interacciones entre agentes durante una negociación

4.6 Diseño detallado

En la arquitectura presentada, se introducen dos nuevos tipos de elementos en el sistema de k8s, el **Scheduler Agent** y el **Node Agent**.

Por un lado, la funcionalidad del Scheduler Agent consiste en:

- Detectar en el etcd a través de un monitor, Pods sin nodo en el que ejecutarse, en estado de Pending y que deban ser planificados por este scheduler.
- Obtener la lista de nodos candidatos para la negociación, para ello consulta con el componente DF de JADE (ver Figura 30).
- Indicar el inicio de la negociación a los nodos candidatos, quedando a la espera de conocer el nodo ganador.
- Una vez conoce el nodo ganador, crea un objeto Binding en el etcd, con toda la información necesaria de la asociación Pod-nodo.

El Scheduler Agent como agente JADE, implementa su funcionalidad mediante un único comportamiento **SchedulerAgentBehaviour**, que es un comportamiento simple (SimpleBehaviour) que permite ser bloqueado para reducir el consumo del nodo maestro de k8s y aumentar así su througput. Este agente se sitúa en el nodo maestro paralelo al kube-scheduler como se muestra en la Figura 33.

Por otro lado, la funcionalidad del Node Agent se divide en dos comportamientos y consiste en:

- En el comportamiento por defecto **NodeAgentBehaviour,** implementa un receptor de mensajes. Aquí, recibe la orden de negociación, con el criterio de negociación y los nodos participantes.
- El otro comportamiento es el de **NodeAgentNegotiation** , donde ejecuta el algoritmo de negociación presentado. Este comportamiento lo activa el NodeAgentBehaviour al recibir la orden de negociación.

El node agent se sitúa en paralelo con el kubelet de cada nodo como se muestra en la Figura 33.

El flujo que sigue un despliegue con el Scheduler Agent como planificador, es similar al del Scheduler. Sin embargo, en lugar de hacer consultas sucesivas al etcd para formar un ranking con los nodos candidatos, los pone a negociar, ver Figura 34.

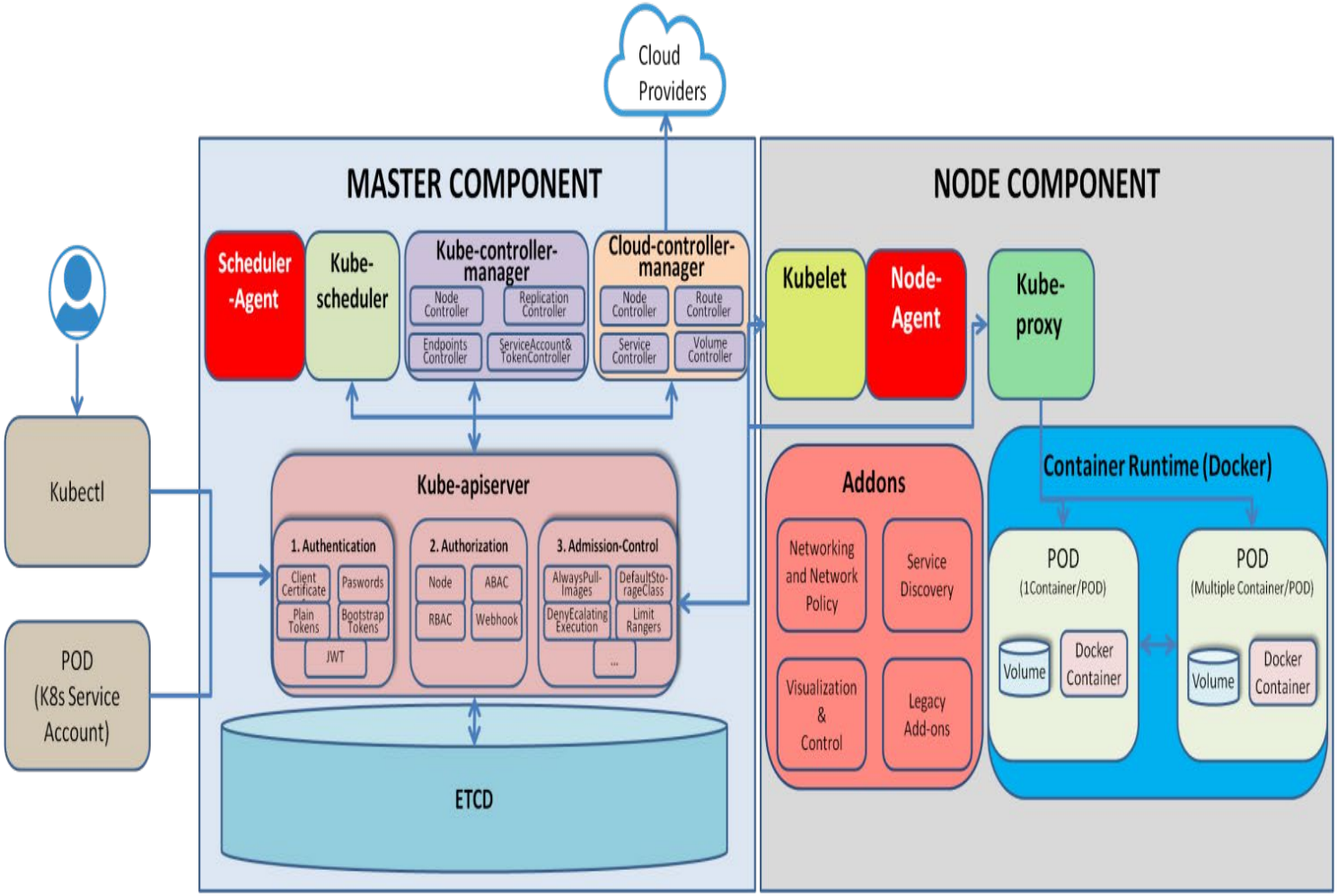


Figura 33 Diseño de extensión de la arquitectura de k8s

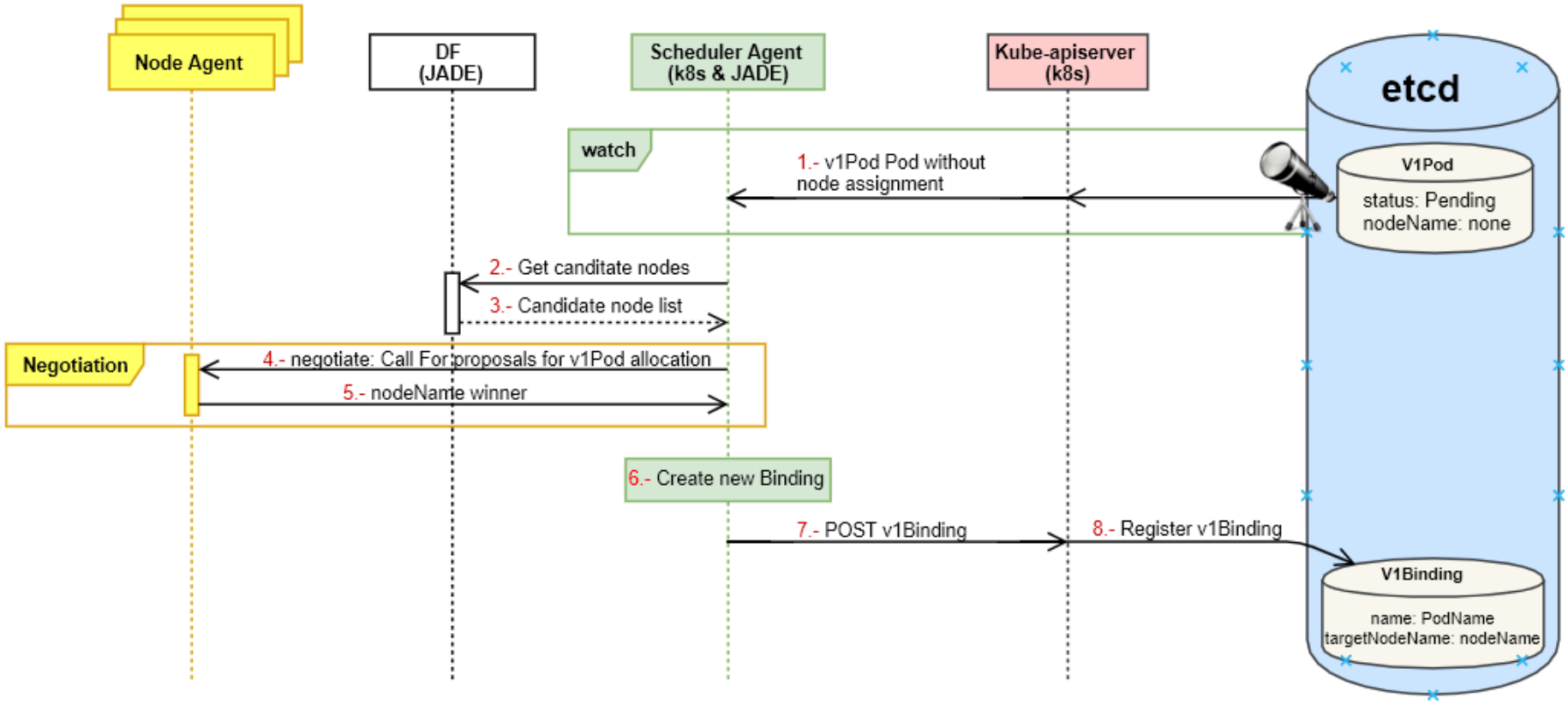


Figura 34 Diagrama de secuencia de la implementación.

5 Pruebas

Una vez presentado el diseño, éste debe ser validado y para ello se realizan una serie de pruebas sobre un clúster k8s. En este apartado, se presenta el clúster k8s sobre el que se van a realizar las pruebas, describiendo tanto el hardware como el software que lo forman. A continuación, se exponen las pruebas realizadas para validar la funcionalidad del diseño, junto con las otras que determinan su rendimiento en cuanto a la respuesta temporal.

5.1 Descripción de clúster:

Este proyecto se realiza en el seno del Departamento de Ingeniería de Sistemas y Automática de la EHU y clúster del que se dispone consta de (se muestran dos imágenes del clúster en la Figura 1**¡Error! No se encuentra el origen de la referencia.**):

- 4 raspberry 3B+
- 1 router
- 1 pantalla
- 1 teclado/ratón
- Varios: hardware necesario para conmutar la entrada de teclado/ratón entre las diferentes rpis (raspberrys), al igual que otro conmutador de salida hdmi con el mismo propósito. Junto a esto, se encuentra el resto de cableado, alimentaciones y seguridades necesarias.

Las rpis tienen **Debian** como OS, y sobre este se ha instalado **k8s**. Como se mencionaba en el Análisis de alternativas la configurabilidad de k8s conlleva que la creación desde cero de un clúster, pueda resultar tediosa. Como solución a esto, k8s ofrece una herramienta llamada **kubeadm**, que facilita el proceso.

Las aplicaciones que ejecutan los nodos (cada rpi) están virtualizadas en contenedores, y sus imágenes guardadas en una nube. Así, cualquier nodo con acceso al API de la nube y perteneciente al clúster k8s (se han unido (join) con el maestro), puede ejecutar aplicaciones. En este caso, se selecciona la nube de google **GCR** (Google Container Registry). Para Poder usarlo, se instala la herramienta **gcloud** en cada nodo. Además, para la comunicación usuario-Api de k8s, se instala la herramienta **kubectl**.

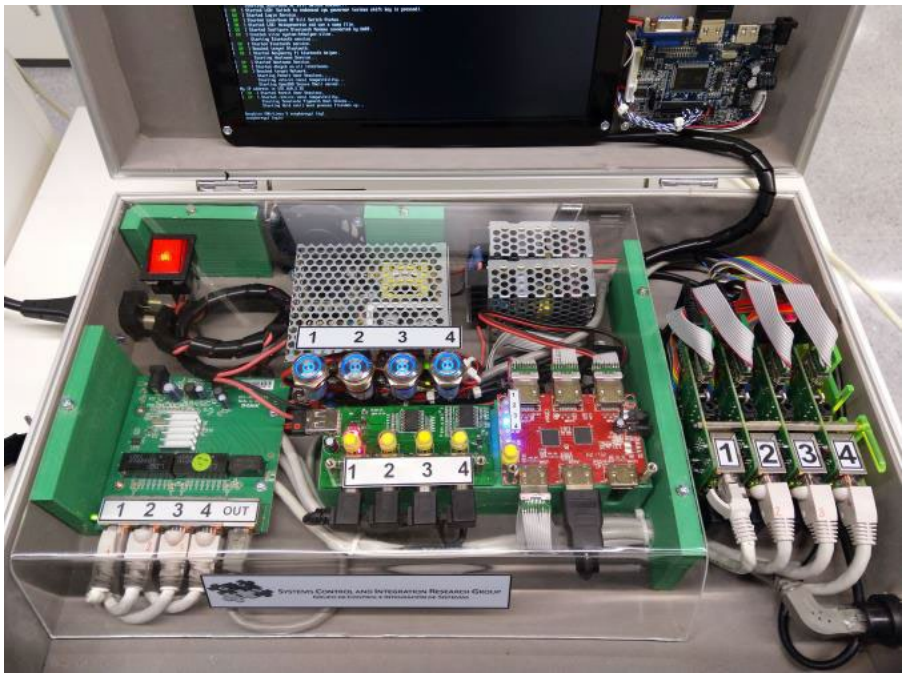


Figura 35 Clúster k8s II

5.2 Pruebas para evaluación cualitativa (funcionalidad)

El uso de la herramienta kubectl, permite al usuario comunicarse con todo el clúster a través del apiserver de k8s. En este caso, se utiliza esta herramienta para realizar una serie de pruebas que permitan analizar si el scheduler propuesto cumple con su función correctamente. Las pruebas se describen en la Tabla 6.

Tabla 6 Pruebas cualitativas

Pruebas cualitativas	
Código	**Descripción**
PCL1	**Planificación de una aplicación**. Se solicita al sistema a través del kubectl el despliegue de una aplicación. Con esta prueba se analiza: 1. El scheduler se comunica con el etcd a través del apiserver para detectar especificaciones de objetos que dan estructura a k8s. 2. Scheduler como parte de k8s se comunica con agentes de nodo para solicitar una negociación distribuida, validando una integración k8s-JADE. 3. Se valida la negociación distribuida, monitorizando el intercambio de mensajes entre agentes de nodo. 4. Scheduler puede crear contenido en la estructura de k8s, al realizar un correcto Binding.
PCL2	**Simulación de fallo de nodo**. Ante un cambio en la disponibilidad del clúster como puede ser la caída de un nodo, se mantiene la coherencia en el scheduling de una aplicación. Con esta prueba se analiza: 1. Correcta integración k8s-JADE, ya que, el scheduler detecta la caída de nodo, por medio de la caída del agente de nodo. 2. Coherencia en la planificación, el nodo caído no entra en negociaciones.
PCL3	**Simulación de un solo nodo en el sistema.** En el peor de los casos en el que solo haya un nodo además del nodo maestro. Con esta prueba se analiza: 1. Correcta planificación, no dándose negociaciones distribuidas.

5.3 Pruebas para evaluación cuantitativa (requisitos temporales)

La distribución realizada en las tareas de planificación dentro de un proceso de despliegue, desahoga al nodo maestro aumentando su throughput. No obstante, se realizan una serie de pruebas que analizan si dicha descentralización afecta sobre los tiempos de respuesta del sistema en el proceso de scheduling. Las pruebas se describen en la Tabla 7.

Tabla 7 Pruebas cuantitativas

Pruebas cuantitativas	
Código	**Descripción**
PCN1	**Planificación de una aplicación, que consta de un Pod**. Se realiza una comparativa temporal en la planificación entre un scheduler centralizado y el desarrollado, repitiendo el proceso 20 veces. Con esta prueba se analiza: 1. Se comprueba la eficiencia de cada scheduler en la planificación de un solo Pod.
PCN2	**Planificación de una aplicación, que consta de 10 réplicas (10Pods).** Se realiza una comparativa temporal en la planificación entre un scheduler centralizado y el desarrollado, repitiendo el proceso 20 veces. Con esta prueba se analiza: 1. El tiempo medio de cada scheduler en la planificación de cada Pod. 2. El tiempo medio en la planificación de todos los Pods.
PCN3	**Planificación de una aplicación, que consta de 100 réplicas (100 Pods).** Se realiza una comparativa temporal en la planificación entre un scheduler centralizado y el desarrollado, repitiendo el proceso 20 veces. Con esta prueba se analiza: 1. El tiempo medio de cada scheduler en la planificación de cada Pod. 2. El tiempo medio en la planificación de todos los Pods.

5.4 Resultados de las pruebas realizadas.

En este apartado se presentan las respuestas de las pruebas realizadas, tanto las funcionales como las temporales.

5.4.1 Resultados de las pruebas cualitativas.

Los resultados obtenidos validan la solución propuesta en este proyecto (ver Tabla 8).

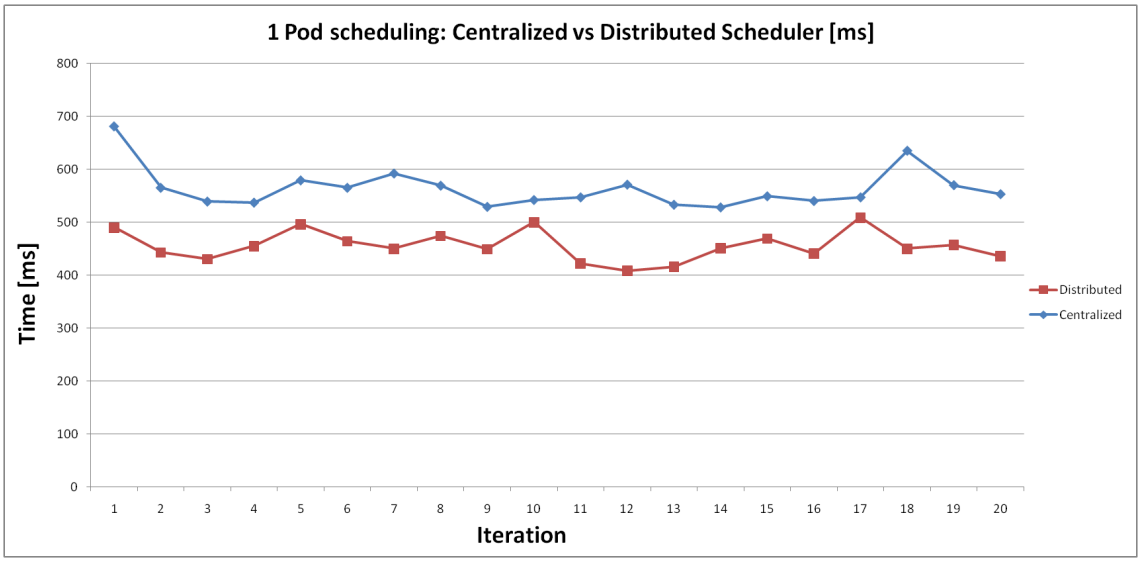
Tabla 8 Resultados de las pruebas cualitativas

Pruebas cualitativas	
Código	**Descripción**
PCL1	OK. Correcta planificación de una aplicación.
PCL2	OK. Correcta planificación de una aplicación.
PCL3	OK. Correcta planificación de una aplicación.

5.4.2 Resultados de las pruebas cuantitativas.

5.4.2.1 PCN1

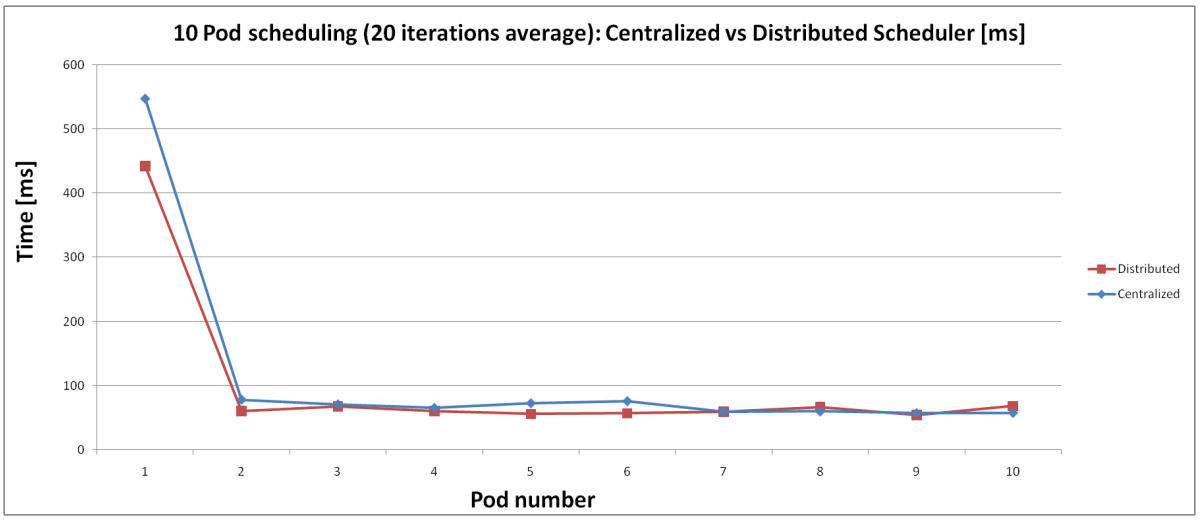
Tras realizar la prueba, se obtiene un tiempo medio en el scheduler centralizado de 562ms y en el distribuido de 455ms. De aquí se concluye, que en bajas tasas de planificación, la distribución responde más rápido. En la Gráfica 1 se muestran las diferencias entre el scheduler centralizado (azul) y el centralizado (rojo). En las gráficas que siguen se mantiene este criterio de colores.



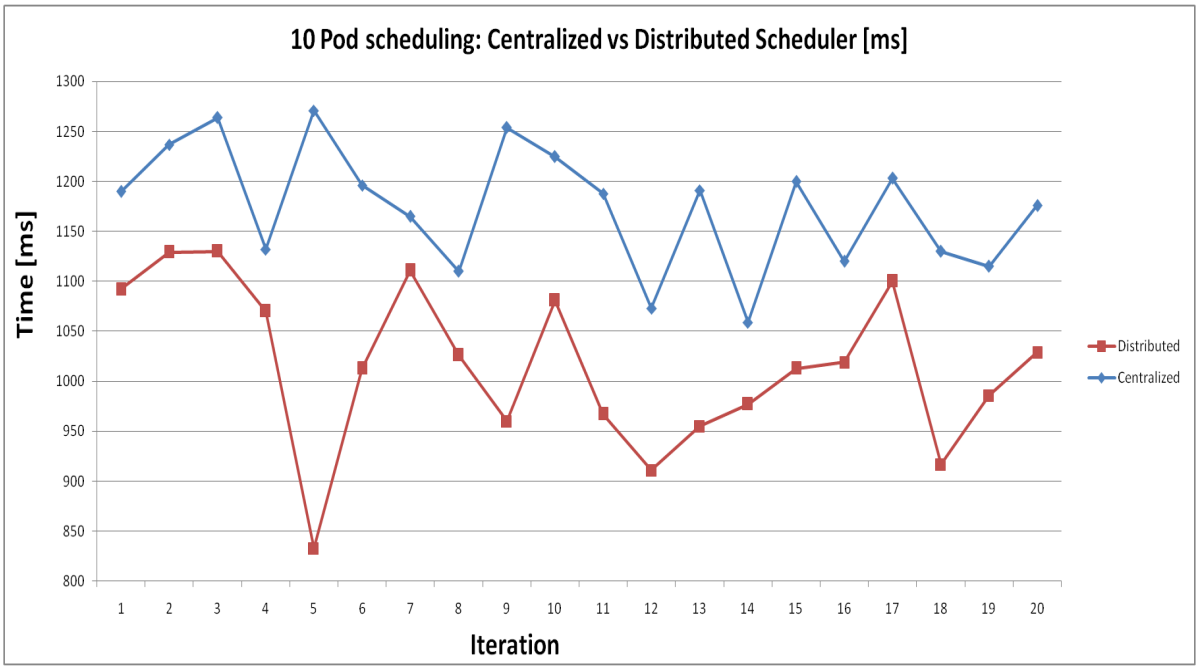
Gráfica 1 Planificación de 1 Pod. Tiempo total de planificación de 1 Pods para cada iteración.

5.4.2.2 PCN2

Tras esta prueba se puede ver cómo el tiempo medio empleado en planificar cada uno de los 10 Pods, es similar (ver Gráfica 2). Sin embargo, el tiempo de cómputo global (duración de la planificación de los 10 Pods), es más corto con el scheduler distribuido (ver Gráfica 3). La media de las 20 iteraciones del scheduler distribuido oscila en torno a los 1012 ms, mientras que el del centralizado lo hace en torno a los 1180.



Gráfica 2 Resultado de la planificación de 10 Pods. Tiempo medio de las 20 iteraciones por cada Pod.



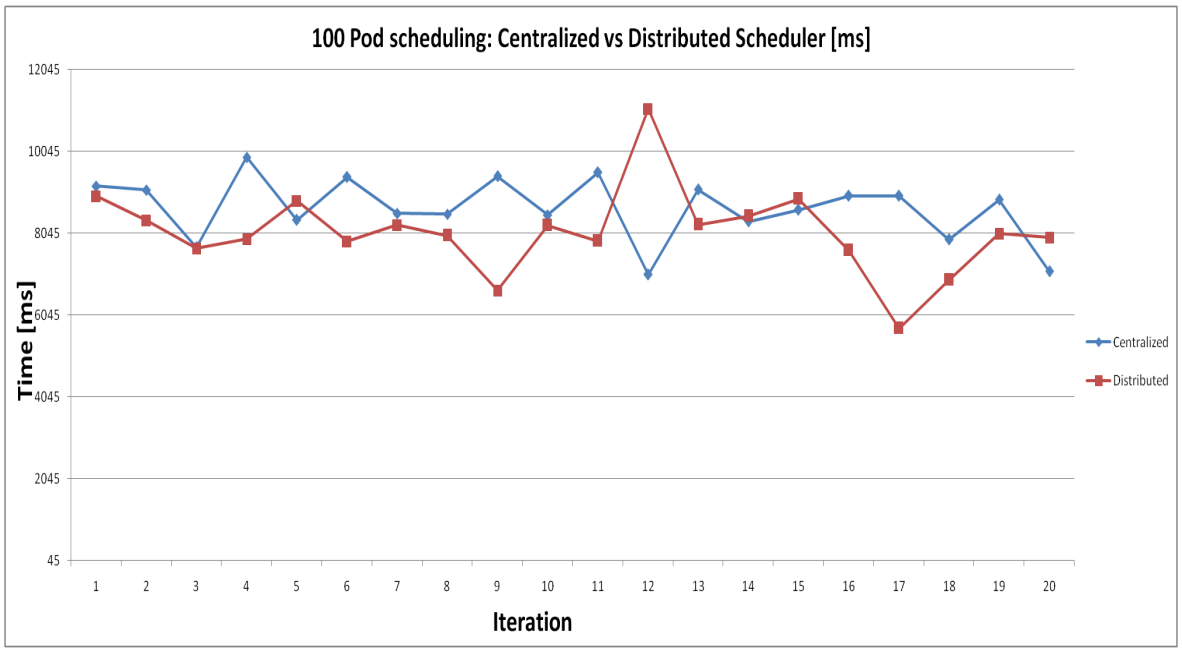
Gráfica 3 Resultado de la planificación de 10 Pods. Tiempo total de planificación de 10 Pods para cada iteración.

5.4.2.3 PCN3

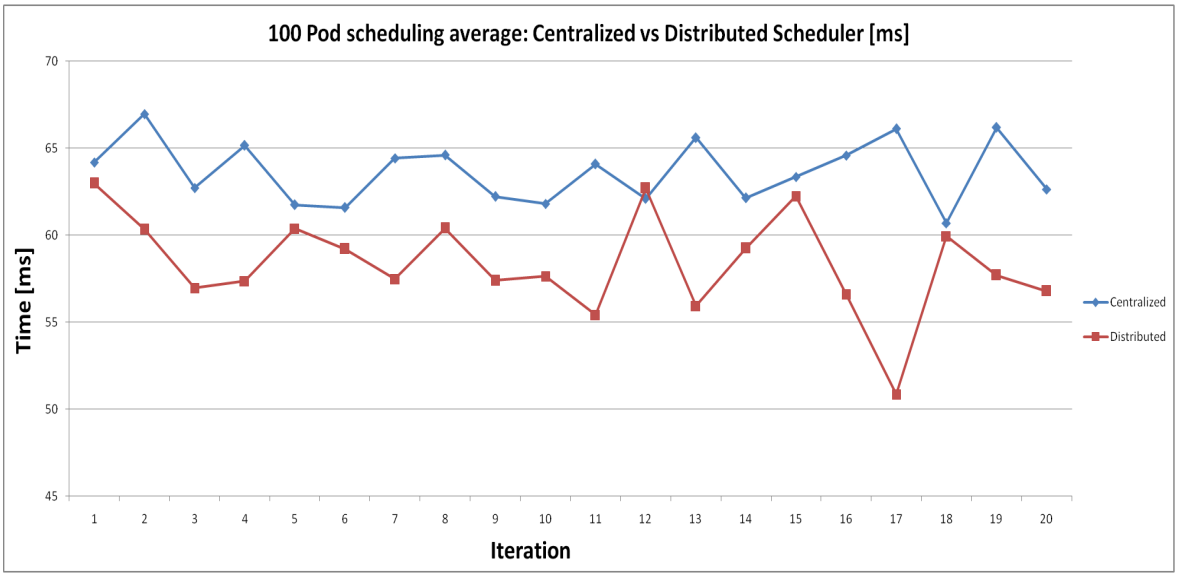
En esta prueba se observa que el tiempo total en la planificación de 100 Pods con el planificador centralizado, es muy similar al distribuido (ver Gráfica 4). Sin embargo, en el tiempo medio en la planificación de 100 Pods se observa que el planificador distribuido es ligeramente más rápido (ver Gráfica 5). En la Gráfica 6 se muestra una comparativa de los tiempos medios por Pod en la planificación de 1, 10, 50 y 100 Pods.

Los tiempos medios presentados requieren de un análisis de la dispersión de los datos con los que han sido obtenidos, para poder concluir si estos resultados se deben al azar. Se realiza un t-test, que se trata de un procedimiento estadístico que permite saber si las diferencias observadas entre dos muestras son generalizables al resto de casos (p.e. a la planificación de 500 Pods). El t-test indica el riesgo que se corre al afirmar que el scheduler agentificado es más rápido que el centralizado, se muestran sus resultados en la Tabla 9.

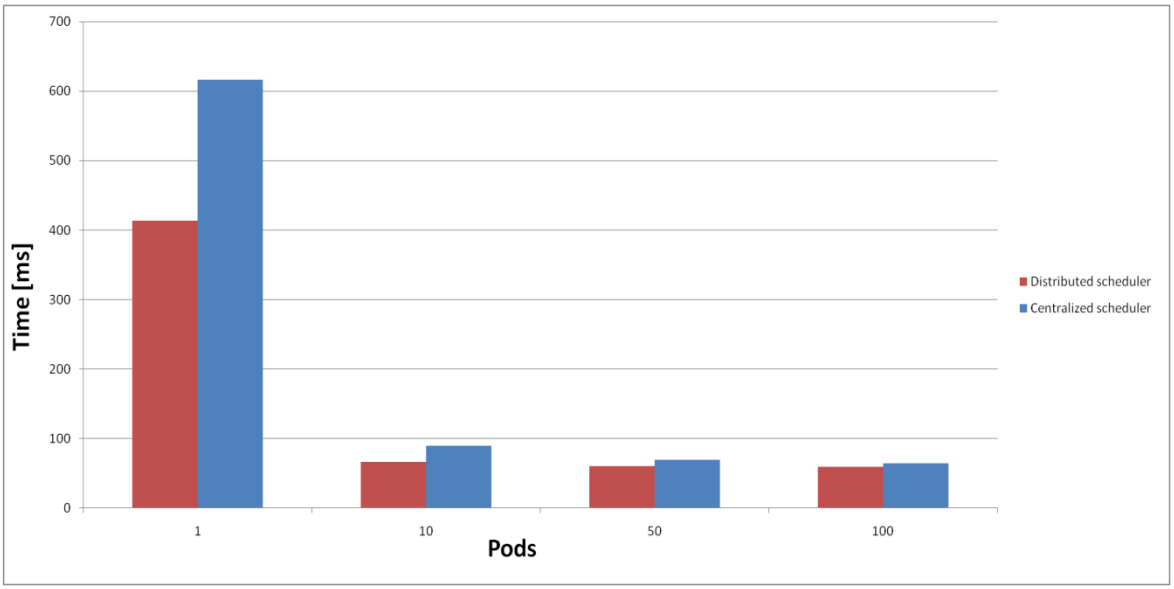
- En el caso de la planificación de un solo Pod, se obtiene un valor de t=10.3>0, pero para saber si esta diferencia se debe al hecho de haber agentificado el scheduler, se obtiene una probabilidad de p<0.001. Esta indica que la probabilidad de equivocarse al afirmar que la agentificación es el factor que produce tiempos de scheduling menores es del 0.1%. Por lo tanto, el scheduler agentificado es más rápido que el centralizado en este caso.
- En el caso de la planificación de 100 Pods se obtiene un valor de t=1.17>0 con una probabilidad de p<0.12. Una probabildidad alta, y no se decir que de forma general el scheduler agentificado vaya a ser siempre más rápido que el centralizado. En este caso, no se indican los asteríscos al lado del valor de t en la Tabla 9.



Gráfica 4 Resultado de la planificación de 100 Pods. Tiempo total en la planificación de 100 pods para cada iteración.



Gráfica 5 Resultado de la planificación de 100 Pods. Tiempo medio de cada iteración en planificar 100 Pods.



Gráfica 6 Resultado del tiempo medio de 20 iteraciones en la planificación de 1 ,10, 50 y 100 Pods.

Tabla 9 Resultados temporales estadísticos (Casquero, Pérez, Sarachaga, & Marcos, 2019)

N pods		Centralized scheduler		Agentified scheduler		t
		M	*SD*	*M*	*SD*	
1		563	37.4	455	27.8	10.3***
10	1st pod	547	25.1	443	36.2	10.5***
	9 replicas	69.7	8.34	63.7	7.97	2.89**
50	1st pod	583	46.4	467	39.4	9.66***
	49 replicas	84.6	9.19	79.6	8.92	0.72
100	1st pod	647	256	415	42.1	3.98***
	99 replicas	80.8	8.99	77.3	8.79	1.17

**p < .01, ***p < .001.

6 METODOLOGÍA

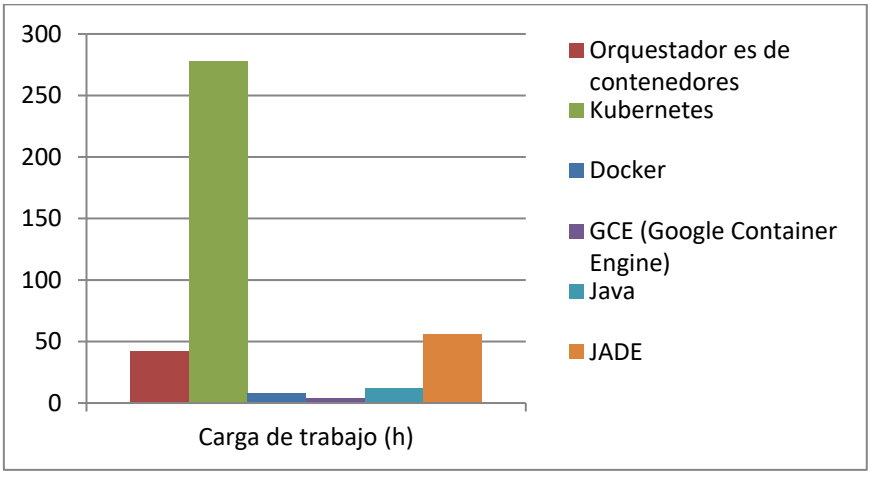
6.1 Planificación de fases y tareas:

6.1.1 Fase 1: Formación

Se inicia el proyecto con un estudio de las funciones de los orquestadores de contenedores. Una vez adquiridos los conceptos básicos, se analizan diferentes alternativas y sus peculiaridades. Se comparan todas las analizadas respecto a la identificación de requisitos que se han identificado. Posteriormente, se analiza en profundidad la alternativa seleccionada: k8s como orquestador y docker como gestor de contenedores. Una vez se conoce lo que es la imagen de un contenedor Docker y cómo crearla, se opta por la herramienta GCE (Google Container Engine) para realizar el pull de las imágenes en k8s. Así mismo, ha habido formación en Java y JADE. Se muestran los tiempos empleados en la Tabla 10 y en la Gráfica 7.

Tabla 10 Formación

Código	Descripción	Duración en días (hábiles)	Carga de horas
F1	Formación	77	400
T1.1	Orquestadores de contenedores	6	42
T1.2	Kubernetes	47	278
T1.3	Docker	2	8
T1.4	GCE (Google Container Engine)	1	4
T1.5	Java	6	12
T1.6	JADE	15	56



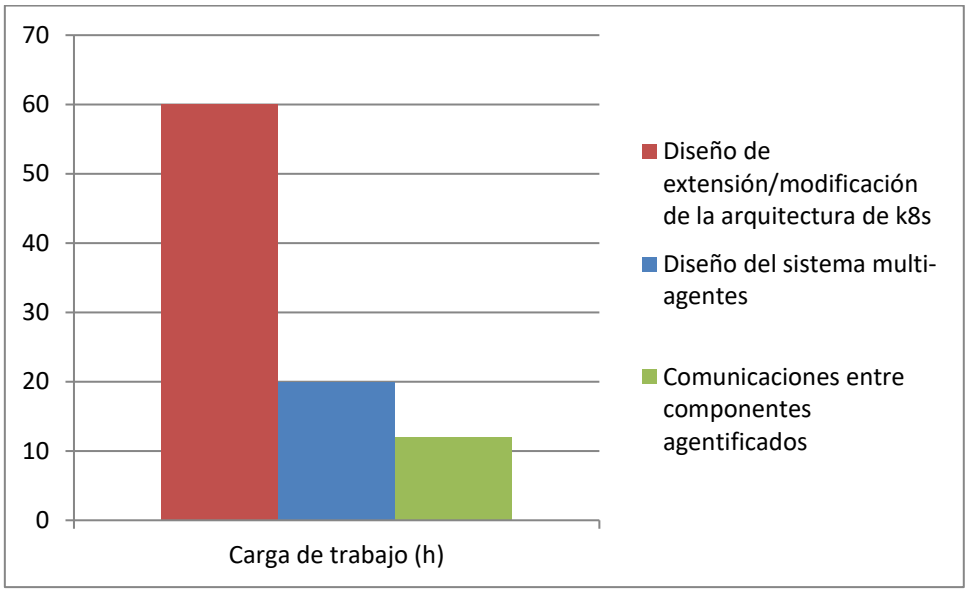
Gráfica 7 Tiempos empleados en la fase de formación

6.1.2 Fase 2: Diseño de extensión/modificación de arquitectura

En esta fase se propone un diseño de extensión de la arquitectura de k8s con componentes agentificados, orientados al despliegue de los contenedores. Para comprender dicho despliegue distribuido, se definen las comunicaciones que se tienen que dar bajo esta nueva arquitectura propuesta. Se muestran los tiempos empleados en la Tabla 11 y en la Gráfica 8.

Tabla 11 Diseño

Código	Descripción	Duración en días (hábiles)	Carga de horas
F2	Diseño	15	92
T2.1	Diseño de extensión/modificación de la arquitectura de k8s	10	60
T2.2	Diseño del sistema multi-agentes	3	20
T2.3	Comunicaciones entre componentes agentificados	2	12



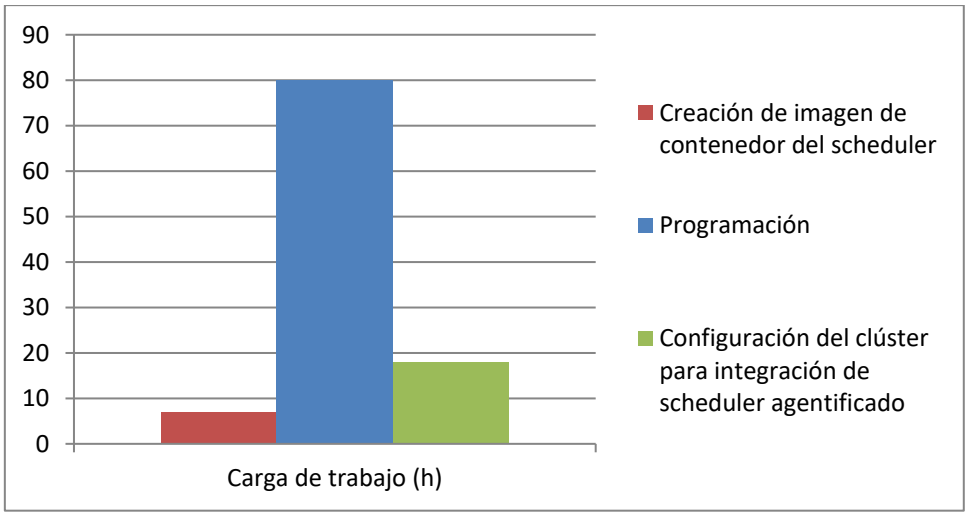
Gráfica 8 Tiempos empleados en la fase de diseño

6.1.3 Fase 3: Implementación del diseño

Una vez se tiene un diseño completo, se realiza una implementación del mismo. En el diseño anterior se propone un nuevo scheduler agentificado en la arquitectura de k8s, junto con un agente por cada nodo. Los componentes k8s ejecutan su funcionalidad en un contenedor. Luego, el primer paso comprende la implementación del nuevo scheduler en un contenedor. Seguido, se proponen las plantillas de los agentes del MAS creado, aportando funcionalidad al scheduler y los nodos. Por último, se configura un clúster de k8s y se le asignan al scheduler creado los permisos necesarios para planificar. Se muestran los tiempos empleados en la Tabla 12 y en la Gráfica 9 Tiempos empleados en la fase de implementación del diseño.

Tabla 12 Implementación

Código	Descripción	Duración en días (hábiles)	Carga de horas
F3	Implementación	16	105
T3.1	Creación de imagen de contenedor del scheduler	1	7
T3.2	Programación	12	80
T3.3	Configuración del clúster para integración de scheduler agentificado	3	18



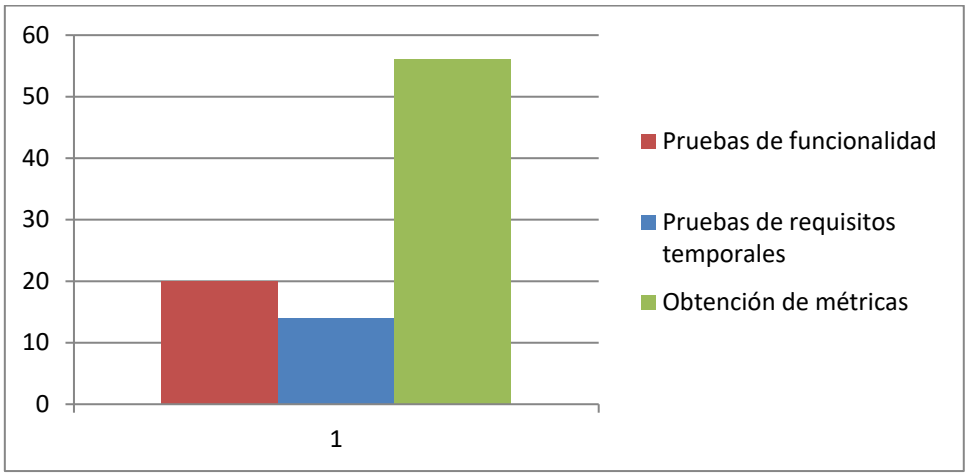
Gráfica 9 Tiempos empleados en la fase de implementación del diseño

6.1.4 Fase 4: Pruebas

La última parte de este proyecto la completa una fase de pruebas, que evalúan el diseño de forma cualitativa y cuantitativa. Además, se obtienen unas gráficas resultado de las pruebas cuantitativas que analizan los requisitos temporales del diseño desarrollado, frente al existente. Se muestran los tiempos empleados en la Tabla 13 y en la Gráfica 10.

Tabla 13 Pruebas

Código	Descripción	Duración en días (hábiles)	Carga de horas
F4	Pruebas cualitativas y cuantitativas	19	90
T4.1	Pruebas de funcionalidad	4	20
T4.2	Pruebas de requisitos temporales	7	14
T4.3	Obtención de métricas	8	56



Gráfica 10 Tiempos empleados en la fase de pruebas

6.2 Diagrama de Gantt

	Título	Fecha de inicio	Fecha de vencimiento	Duración
1	**Orquestadores para la...**	09/01/2019	29/05/2019	
2	**F1 Formación**	09/01/2019	05/04/2019	
3	T1.1 Orquestad...	09/01/2019	16/01/2019	6d
4	T1.2 Kubernetes	17/01/2019	22/03/2019	47d
5	T1.3 Docker	05/03/2019	06/03/2019	2d
6	T1.4 GCE (Googl...	07/03/2019	07/03/2019	1d
7	T1.5 Java	08/03/2019	15/03/2019	6d
8	T1.6 JADE	18/03/2019	05/04/2019	15d
9	**F2 Diseño**	25/03/2019	17/04/2019	
10	T2.2 Diseño del ...	25/03/2019	12/04/2019	15d
11	T2.1 Diseño de ...	25/03/2019	27/03/2019	3d
12	T2.3 Comunicac...	15/04/2019	17/04/2019	3d
13	**F3 Implementación**	18/04/2019	09/05/2019	
14	T3.1 Creación d...	18/04/2019	18/04/2019	1d
15	T3.2 Programaci...	19/04/2019	06/05/2019	12d
16	T3.3 Configuraci...	07/05/2019	09/05/2019	3d
17	**F4 Pruebas cualitat...**	10/05/2019	29/05/2019	
18	T4.1 Pruebas de...	10/05/2019	15/05/2019	4d
19	T4.2 Pruebas de...	16/05/2019	24/05/2019	7d
20	T4.3 Obtención ...	20/05/2019	29/05/2019	8d

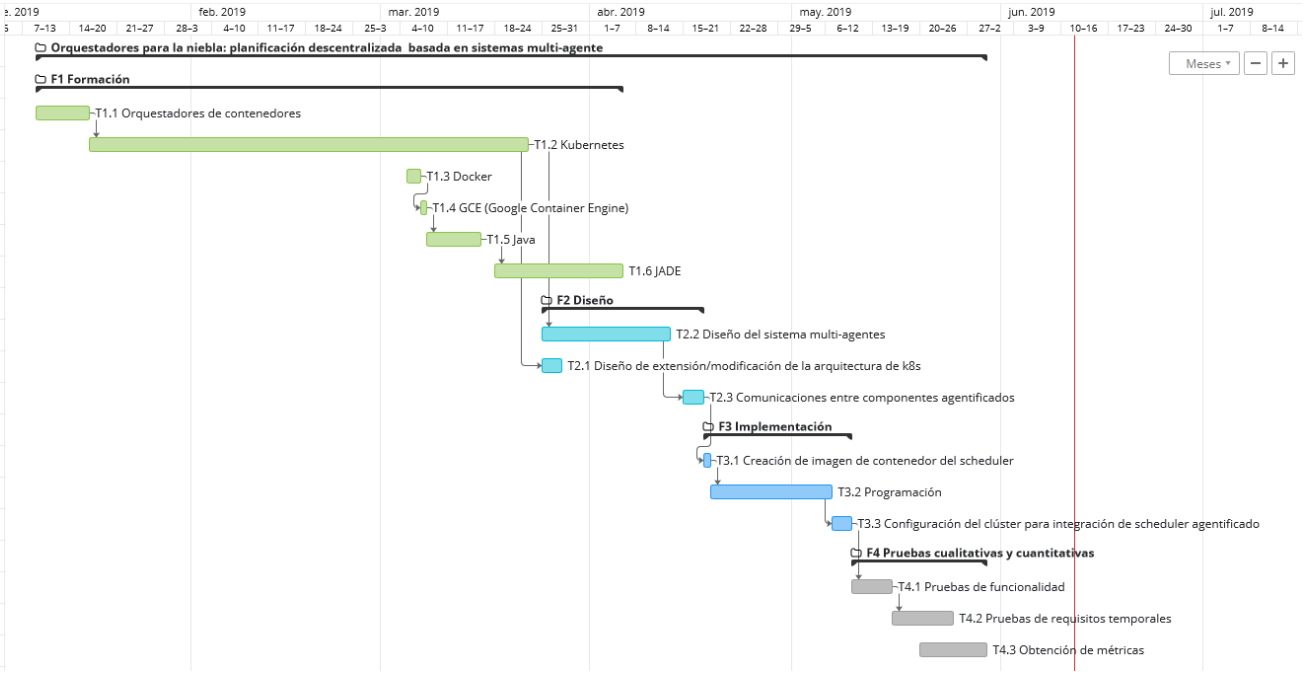


Figura 36 Diagrama de Gantt

6.3 Aspectos Económicos

En este apartado se plasman los por menores económicos que acarrea este proyecto. Por un lado, las horas invertidas por parte de las personas físicas (ver Tabla 14). Por otro lado, las amortizaciones de los materiales empleados (ver Tabla 15). Finalmente, se muestra un resumen de los gastos totales (ver Tabla 16).

Tabla 14 Horas invertidas por personal

Horas invertidas				
Trabajador	**Cantidad**	**Coste horario (€)**	**Horas invertidas**	**Coste total(€)**
Ingeniero	1	15	660	9900
Profesor Ingeniero	1	25	200	5000
Directora del proyecto	1	25	100	2500
Total				17400

Tabla 15 Amortizaciones

	Amortizaciones					
	Concepto	**Cantidad**	**Coste de cada unidad(€)**	**Vida útil (años)**	**Uso (h)**	**Coste (€)**
	Ordenador	1	1200	4	660	113,66
Clúster	Rpi 3 B+	4	45	4	240	6,20
	Tarjeta MicroSD	4	20	4	240	2,76
	Selector de fuente USB	1	15	4	240	0,52
	Selector de fuente HDMI	1	40	4	240	1,38
	Router	1	80	4	240	2,76
	Maletín	1	40	4	240	1,38
	Pantalla	1	70	4	240	2,41
	Varios		100	4	240	3,44
	Total					134,50

Tabla 16 Resumen de gastos

Computo Global	
Concepto	**Coste (€)**
Horas invertidas	17400,00
Amortizaciones	134,50
Subtotal	17534,50
Costes indirectos (2%)	350,69
Total	17885,19

7 Conclusiones y trabajo futuro.

7.1 Conclusiones

El trabajo realizado en este proyecto ha establecido una base teórica ordenada de la arquitectura del orquestador de contenedores k8s. Con el conocimiento adquirido en este trabajo, junto con el resto de conocimientos adquiridos a lo largo del Máster en Ingeniería Industrial, se ha conseguido cumplir con el objetivo principal del proyecto. Es decir, se ha descentralizado el proceso de despliegue de un orquestador de contenedores, mediante una negociación basada en sistemas multi-agentes (MAS).

El diseño propuesto ha supuesto la integración de JADE en k8s y se ha validado su funcionamiento con diferentes pruebas, sobre un clúster desarrollado por el grupo GSIC. Además, se han realizado una serie de pruebas temporales con las que se ha demostrado que el planificador propuesto es más rápido que el ofrece k8s, en el despliegue de un Pod. Sin embargo, en el despliegue de más Pods, aunque se han obtenido resultados positivos que muestran mejoras temporales, no se puede asegurar que esto vaya a ser siempre así. Por lo que, conviene en un futuro realizar pruebas más completas que traten de analizar el rendimiento de la propuesta.

Este desarrollo, establece una base para componer una arquitectura de niebla en la que se pueden alojar aplicaciones sensibles al contexto. No solo eso, se ha diseñado una solución que permite personalizar criterios de negociación, en función de los cuales, los agentes de nodo pueden llegar a ejecutar diferentes algoritmos, que permitan reconfiguraciones dinámicas de estas aplicaciones. Esta es una de las líneas de investigación que abre el trabajo presentado en este documento, que compete la integración del middleware del GCIS MASRECON con k8s, ya que este middleware extiende la plataforma JADE.

7.2 Líneas de desarrollo futuro.

El proyecto desarrollado abre la posibilidad de descentralizar procesos de la gestión de contenedores, mediante la agentificación de diferentes componentes del orquestador. Una línea interesante de desarrollo que permite crear una arquitectura de la niebla consiste en una integración del middleware desarrollado por el grupo GCIS en k8s. Para una completa integración de middleware, se deben agentificar los siguientes componentes de la arquitectura de k8s:

- **Kubelet**: la agentificación de este componente supone una considerable mejora del proyecto presentado en este documento, ya que permite al sistema MAS conocer el estado de todos los nodos. No solo esto, la plantilla del agente de nodo creada se puede usar como base en la agentificación de este componente para crear nuevos criterios y algoritmos de negociación que permitan alojar aplicaciones reconfigurables dinámicas. Además, mejora la cohesión entre k8s y el MAS ya que los nodos son representados en ambos sistemas por el mismo elemento, evitando una posible duplicidad de datos.
- **Pod**: la agentificación de las aplicaciones supone una mejora en la descentralización de procesos, permitiendo a k8s beneficiarse de las ventajas que ofrecen los agentes JADE.

8 Bibliografía

Armentia, A., Gangoiti, U., Priego, R., Estévez, E., & Marcos, M. (2015). Flexibility support for homecare applications based on models and multi-agent technology. *Sensors (Switzerland)*, *15*(12), 31939–31964. https://doi.org/10.3390/s151229899

Bellavista, P., & Zanni, A. (2017). Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi. *Proceedings of the 18th International Conference on Distributed Computing and Networking - ICDCN '17*, 1–10. https://doi.org/10.1145/3007748.3007777

Bonomi, F., Milito, R., Zhu, J., & Addepalli, S. (2012). Fog computing and its role in the internet of things. *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing - MCC '12*, 13. https://doi.org/10.1145/2342509.2342513

Casquero, O., Pérez, F., Sarachaga, I., & Marcos, M. (2019). *Distributed scheduling in Kubernetes based on MAS for Fog-in-the-loop applications*. 6–9.

Chopra, R. (2019). GitHub - coreos/flannel: flannel is a network fabric for containers, designed for Kubernetes. Retrieved June 8, 2019, from https://github.com/coreos/flannel

GARLAN, D., & SHAW, M. (2010). *AN INTRODUCTION TO SOFTWARE ARCHITECTURE*. https://doi.org/10.1142/9789812798039_0001

Hashicorp. (2019). Building Resilient Infrastructure with Nomad: Data Safety and Outage Recovery. Retrieved May 24, 2019, from https://www.hashicorp.com/blog/resilient-infrastructure-with-nomad-fault-tolerance-outage-recovery

HashiCorp. (2019). Nomad by HashiCorp. Retrieved May 24, 2019, from https://www.nomadproject.io/intro/index.html

Hesselbach Serra, X., & Altés Bosch, J. (2002). Análisis de redes y sistemas de comunicaciones. *Univ. Politèc. de Catalunya*, 186. Retrieved from https://books.google.com.pe/books?id=11DSMYYKvL0C&pg=PA89&dq=cadena+de+markov+en+tiempo+discreto&hl=es&sa=X&ved=0ahUKEwi-1IHHjMXXAhUEwiYKHfV-A60Q6AEIJDAA#v=onepage&q=cadena de markov en tiempo discreto&f=false

Hoque, S., Brito, M. S. de, Willner, A., Keil, O., & Magedanz, T. (2017). Towards Container Orchestration in Fog Computing Infrastructures. *Proceedings - International Computer Software and Applications Conference*, *2*, 294–299. https://doi.org/10.1109/COMPSAC.2017.248

Huvio, E., Grönvall, J., & Främling, K. (2002). Tracking and tracing parcels using a distributed computing approach. *Proceedings of the 14th Annual Conference for Nordic Researchers in Logistics (NOFOMA'2002)*, 29–43. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=B94D565000AAD764A2

D1ABD15FD7435B?doi=10.1.1.59.8340&rep=rep1&type=pdf

Isaac Eldridge. (2017). What is Container. Retrieved May 24, 2019, from https://blog.newrelic.com/engineering/container-orchestration-explained/

Kagermann, H., Wolf-Dieter, L., & Wolfgang, W. (2011). Industrie 4.0: Mit dem Internet der Dinge auf dem Weg zur 4. industriellen Revolution [From Internet of Thing to the fourth industry revolution]. *VDI Nachrichten*, (13), 3–4. https://doi.org/10.13140/RG.2.1.1205.8966

Khakimov, A., Muthanna, A., & Muthanna, M. S. A. (2018). Study of fog computing structure. *Proceedings of the 2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, ElConRus 2018*, *2018-Janua*, 51–54. https://doi.org/10.1109/EIConRus.2018.8317028

Kubernetes. (2018). Configuring kubelet Garbage Collection - Kubernetes. Retrieved May 25, 2019, from November 02, 2018 website: https://kubernetes.io/docs/concepts/cluster-administration/kubelet-garbage-collection/

Kubernetes. (2019a). Assigning Pods to Nodes - Kubernetes. Retrieved May 25, 2019, from https://kubernetes.io/docs/concepts/configuration/assign-pod-node/

Kubernetes. (2019b). Cluster Networking | Kubernetes. Retrieved May 25, 2019, from April 15, 2019 website: https://kubernetes.io/docs/concepts/cluster-administration/networking/

Kubernetes. (2019c). community/resource-management.md at master · kubernetes/community · GitHub. Retrieved May 25, 2019, from https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture/resource-management.md

Kubernetes. (2019d). Configure Out Of Resource Handling - Kubernetes. Retrieved May 25, 2019, from March 20, 2019 website: https://kubernetes.io/docs/tasks/administer-cluster/out-of-resource/

Kubernetes. (2019e). Kubernetes Components - Kubernetes. Retrieved May 25, 2019, from https://kubernetes.io/docs/concepts/overview/components/#node-components

Kubernetes. (2019f). Kubernetes Documentation - Kubernetes. Retrieved May 24, 2019, from https://kubernetes.io/docs/home/

Kubernetes. (2019g). Master-Node communication - Kubernetes. Retrieved May 25, 2019, from February 28, 2019 website: https://kubernetes.io/docs/concepts/architecture/master-node-communication/

Kubernetes. (2019h). Taints and Tolerations - Kubernetes. Retrieved May 25, 2019, from https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/

Kubernetes. (2019i). What is Kubernetes? - Kubernetes. Retrieved May 24, 2019, from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#why-containers

Leitão, P. (2009). Agent-based distributed manufacturing control: A state-of-the-art survey. *Engineering Applications of Artificial Intelligence*, *22*(7), 979–991. https://doi.org/10.1016/j.engappai.2008.09.005

Loayza, A., Proaño, R., & Camacho, D. O. (2013). *Aplicaciones sensibles al contexto. Tendencias actuales. (Current trends in context-aware applications.)*. (2), 95–110. Retrieved from http://ingenieria.ute.edu.ec/enfoqueute/

Marhubi, K. (2015). What even is a kubelet? Retrieved May 25, 2019, from Aug 27, 2015 website: http://kamalmarhubi.com/blog/2015/08/27/what-even-is-a-kubelet/

Mesos. (2019). Marathon: Install Marathon. Retrieved May 24, 2019, from https://mesosphere.github.io/marathon/docs/

Nwana, H. S. (1996). Software agents: an overview. *The Knowledge Engineering Review*, *11*(3), 205–244. https://doi.org/10.1017/s026988890000789x

Paul Morie. (2017). community/api-conventions.md at master · kubernetes/community · GitHub. Retrieved May 25, 2019, from 3/7/2017 website: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#verbs-on-resources

Rajdeep Dua. (2017). *一种适用于Docker Swarm集群的调度策略和算法*. 1–5. Retrieved from https://www.slideshare.net/rajdeep/docker-swarm-introduction

Rancher labs. (2019). k3s - Lightweight Kubernetes | k3s. Retrieved May 24, 2019, from https://k3s.io/

Rufino, J., Alam, M., Ferreira, J., Rehman, A., & Tsang, K. F. (2017). Orchestration of containerized microservices for IIoT using Docker. *Proceedings of the IEEE International Conference on Industrial Technology*, 1532–1536. https://doi.org/10.1109/ICIT.2017.7915594

Shah, A. A., Piro, G., Grieco, L. A., & Boggia, G. (n.d.). *A Qualitative Cross-Comparison of Emerging Technologies for Software-Defined Systems*. Retrieved from https://linuxcontainers.org/lxd/

Swarm features Docker. (2019). Swarm mode overview | Docker Documentation. Retrieved May 24, 2019, from https://docs.docker.com/engine/swarm/

Tua, E. (2019). community/architecture.md at master · kubernetes/community · GitHub. Retrieved May 25, 2019, from https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture/architecture.md

Wang, S., Wan, J., Li, D., & Zhang, C. (2016). Implementing Smart Factory of

Industrie 4.0: An Outlook. *International Journal of Distributed Sensor Networks*, *2016*(1), 3159805. https://doi.org/10.1155/2016/3159805

Zhang, H. H. (2017). Kubernetes Architecture - beyond a black box - Part 2. Retrieved May 25, 2019, from https://www.slideshare.net/harryzhang735?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview

Zhou, K., Liu, T., & Zhou, L. (2016). Industry 4.0: Towards future industrial opportunities and challenges. *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery, FSKD 2015*, 2147–2152. https://doi.org/10.1109/FSKD.2015.7382284