Sistema distribuido para la ejecución controlada de código arbitrario

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

Realizado por:
Jonathan Sánchez Paredes
Alberto Velázquez Alonso

Director:
Luis Llana
Codirector:
Enrique Martín Martín
Departamento de Sistemas Informáticos y Computación

Facultad de Informática
Universidad Complutense de Madrid

Madrid, junio 2017

Autorización de difusión

viernes, 16 de junio de 2017

Los abajo firmantes, alumnos y tutor del Trabajo Fin de Grado (TFG) en el Grado en Ingeniería Informática de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Grado (TFG): "Sistema distribuido para la ejecución controlada de código arbitrario", realizado durante el curso académico 2016-2017 bajo la dirección de Luis Llana y la codirección de Enrique Martín en el Departamento de sistemas informáticos y computación. Así mismo autorizan a la Universidad Complutense de Madrid a que sea depositado en acceso abierto en el repositorio institucional con el objeto de incrementar la difusión, uso e impacto del TFG en Internet y garantizar su preservación y acceso a largo plazo.

Jonathan Sánchez Paredes

Alberto Velázquez Alonso

Agradecimientos

Queremos agradecer de una manera especial a nuestro director Luis Llana y a nuestro codirector Enrique Martín, por habernos elegido para su proyecto, confiando en nosotros y facilitándonos un Clúster [11] para el desarrollo del proyecto.

Por otro lado, también queremos agradece a nuestros familiares su implicación y estar en todo momento a nuestro lado, escuchando y dando su opinión a los problemas que nos iban surgiendo.

Por último, agradecer a nuestros compañeros y animar con sus respectivos trabajos de fin de grado, estamos muy orgullosos de haber compartido este camino con vosotros aun sabiendo que no será el último camino que recorramos juntos.

Prólogo

Todo comenzó en junio del 2016 tras la finalización de los exámenes, cuando nos planteamos la opción de coger un trabajo de fin de grado para el 2016/2017. Tras surgirnos varias ideas y dar varias vueltas a la lista de trabajos propuestos, nos topamos con la distribución de sistemas, el cual nos llamó mucho la atención ya que son tecnologías con muchas salidas laborales y en el grado apenas tenemos contacto con ellas. Siendo tecnologías con muy poco recorrido, es decir, son relativamente nuevas, por ejemplo, *Kubernetes* fue lanzado en junio del 2014.

Estos fueron los principales alicientes para que nos pusiéramos en contacto con Luis para iniciar este trabajo de fin de grado juntos. Tras una primera reunión para una toma de contacto, nos comunican que el sistema distribuido es para reemplazar en un futuro la arquitectura de la aplicación Juez de la facultad de informática.

Resumen

La motivación original de este proyecto es la mitigación de riesgos en el uso de un juez virtual, que es una aplicación que facilita el envío, ejecución y evaluación de ejercicios prácticos vía Internet. Los jueces virtuales más populares en entornos educativos son FLOP [27], DOMJudge, Mooshak y AceptaElReto.

Este tipo de aplicaciones compilan y ejecutan código arbitrario, y por tanto precisan un entorno de ejecución seguro, aislado y en el que pueda limitarse el uso de recursos como memoria o tiempo de CPU. Además, dado que los alumnos pueden enviar sus prácticas en cualquier momento del día, estos sistemas necesitan alta disponibilidad.

Este proyecto pretende dar respuesta a estas necesidades creando un entorno de ejecución seguro, controlado y de alta disponibilidad para la ejecución remota de código arbitrario. Para ello recogeremos los requisitos de la aplicación y analizaremos las distintas alternativas a partir de sus características y adecuación a los requisitos. Finalmente desarrollaremos una aplicación que dé solución a las necesidades indicadas basándonos en los requisitos y en los resultados de nuestra investigación.

Palabras clave

Docker
Kubernetes
Celery
Redis
RabbitMQ
TaskQueue
Orquestación
Clúster
Sistema Distribuido

Abstract

The original motivation for this project is risk mitigation for the use of a virtual judge, which is an application that facilitates sending, running and grading practical assignments online. Some popular virtual judges in educational environments are FLOP, DOMJudge, Mooshak and AceptaElReto.

This kind of applications compile and execute arbitrary code, so they require a running environment that is secure, isolated and that can limit resource usage such as memory or CPU time. Also, given that students can send their assignments at any time, these systems need high availability.

This project intends to give an answer to these needs by providing a secure, controlled and high availability running environment for the remote execution of arbitrary code. To this end we will gather this application's requirements and we will analyze the different alternatives based on their features and adherence to requirements. Finally, we will develop an application that can solve the stated needs, based on the requirements and the results of our research.

Keywords

Docker
Kubernetes
Celery
Redis
RabbitMQ
TaskQueue
Orchestration
Cluster
Distributed System

Índice

1. Introducción

La computación desde sus principios ha pasado por muchos cambios, siendo pioneros los grandes computadores que eran capaces de realizar innumerables tareas, pero con gran coste tanto en recursos como en consumo. En cambio, con el paso del tiempo los ordenadores personales han ido cogiendo protagonismo y a día de hoy son capaces de mejorar a los grandes computadores. De esta evolución surgen los denominados sistemas distribuidos, donde varios computadores pequeños trabajan concurrentemente, comunicándose para lograr un objetivo. También, aplicando modularidad a la aplicación separando el interfaz de la ejecución.

Este avance también lo quiere trasladar la Facultad de Informática de la Universidad Complutense de Madrid, trasladando la aplicación del Juez, en concreto FLOP, desde un único *host* [22], a un sistema distribuido.

Los jueces son utilizados principalmente en la docencia, este software facilita a los profesores poner ejercicios a sus alumnos, los alumnos desarrollan el código del ejercicio y lo envían para su corrección. El juez compila el programa y devuelve al alumno el resultado de la compilación, a su vez, deja almacenado la nota del alumno. Algunos jueces automáticos son FLOP, DOMJudge, Mooshak, AceptaElReto.

FLOP

Es un software que sigue la metodología Test Driven Design, el uso de este programa está destinado a la programación y el aprendizaje. Este software alberga problemas de programación, permite al profesor añadir fácilmente otros nuevos y también evalúa automáticamente las soluciones enviadas por los alumnos.

1.1. Motivación

En los últimos años se han producido avances en cuanto a la arquitectura de los sistemas, trasladándose hacia la distribución con el fin de repartir la carga en los servidores, o incluso para consumir solo los recursos necesarios de un servicio y pagar solo por ellos, no por toda la máquina. Como el juez automático FLOP, está ejecutándose en un único host, se quiere trasladar a un sistema distribuido.

Un sistema distribuido es un conjunto de computadores separados físicamente, pero conectados por una red. El usuario del sistema percibe esta distribución desde el exterior como un único sistema.

Para facilitar la distribución y la ejecución de aplicaciones dentro del sistema distribuido se utiliza *Docker*, y para orquestarlo [15] se utilizarán distintas alternativas software, como son Kubernetes, Docker Swarm y Mesos.

La idea principal del trabajo es desarrollar un sistema de ejecución para que cualquier juez automático lo pueda usar, incluyendo a FLOP.

1.2. Objetivos

El objetivo principal del proyecto es implementar un sistema de ejecución remota encargado de repartir la carga entre varios nodos [22] y conseguir una eficiencia mayor, consiguiendo así satisfacer un mayor número de peticiones. Al estar compuesto por varios computadores, estos no tienen por qué tener una alta computación, por lo que reducimos el coste de nuestro sistema. También, disponemos de un sistema escalable, por sus módulos y disponible ya que replicamos recursos.

En algunas situaciones es necesario ejecutar código externo del que no se tienen garantías de su funcionamiento, por lo que se suelen utilizar entornos controlados como máquinas virtuales o contenedores [16] software (Docker). Cuando el número de ejecuciones es bastante elevado, es imprescindible contar con un sistema distribuido que gestione distintas máquinas y balancee la carga para obtener una eficiencia óptima. El objetivo de este proyecto es desarrollar un sistema distribuido para la ejecución segura de código no confiable, proporcionando una interfaz sencilla para su gestión.

1.3. Estado del arte

Ante la búsqueda de aplicaciones que tengan el mismo objetivo, compilar código arbitrario seguro, se han encontrado bastantes aplicaciones que realizan las mismas funciones, entre ellas están http://fiddles.io, http://ideone.com y http://codepad.org.

Dichas páginas ofrecen el mismo servicio, pero no conocemos su funcionamiento interno, pueden tener un único host o funcionar con un sistema distribuido. Por falta de información se desconoce el estado de esta tecnología.

Visitando las páginas oficiales de Docker, Kubernetes, Docker Swarm, se puede ver que los sistemas distribuidos orquestados están creciendo notablemente en cuanto a número de usuarios, siendo una tecnología que está creciendo rápidamente.

VIRTUALIZACIÓN

La virtualización consiste en la simulación de un computador dentro de otro mediante software, opcionalmente con la ayuda de dispositivos especiales de hardware. La máquina virtual se muestra como una máquina real ante los programas que ejecuta.

CONTENEDOR

Un contenedor es un entorno aislado en el cual los procesos que se ejecutan están separados del resto de procesos de la máquina. El sistema de archivos visible para los procesos del contenedor también es independiente.

DOCKER

Docker es la tecnología de contenedores más popular actualmente. Docker depende de varias características del núcleo de Linux, y aunque corre en otros sistemas operativos, lo hace en una máquina virtual.

RKT

RKT es otra tecnología de contenedores desarrollada por el equipo de CoreOS ideada para funcionar con su sistema operativo, y orientada a la seguridad.

ORQUESTACIÓN

La orquestación es la gestión automatizada de contenedores, incluyendo el arranque, parada y reinicio de éstos. Dichos contenedores pueden encontrarse en una sola máquina o en varias, dependiendo de las características del software de orquestación utilizado.

KUBERNETES

Kubernetes es un sistema de orquestación de contenedores desarrollado por Google diseñado para operar con Docker y rkt [26].

DOCKER SWARM

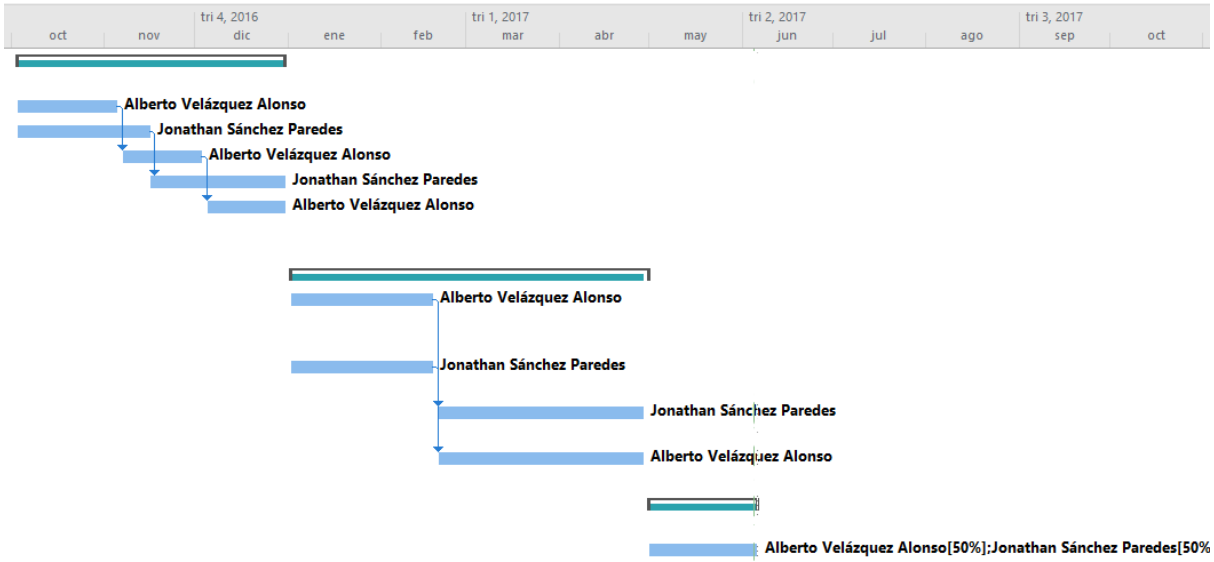
Docker Swarm es un sistema de orquestación desarrollado por Docker, y que por tanto sólo funciona con contenedores Docker. Su meta es utilizar la misma API existente en Docker para automatizar la gestión de múltiples contenedores.

APACHE MESOS

Mesos es un software desarrollado para automatizar la gestión de clústeres y que permite su uso para la orquestación de contenedores.

1.4. Plan de trabajo

La primera fase se basa en una toma de contacto e investigación de las tecnologías, que se pueden utilizar para el sistema. Por último, decidir qué tecnologías se usarán. En la segunda fase se tratan temas más prácticos, llevando a cabo la instalación de las tecnologías que se van a utilizar. La última fase en la que se redacta la memoria, trasladando los resultados obtenidos.



	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Prede	Nombres de los recursos
1		**Fase 1 - Investigación**	**65 días**	**lun 03/10/16**	**vie 30/12/16**		
2		Uso de Docker	25 días	lun 03/10/16	vie 04/11/16		Alberto Velázquez Alonso
3		Web Services	32 días	lun 03/10/16	mar 15/11/16		Jonathan Sánchez Paredes
4		Colas de mensajes	20 días	lun 07/11/16	vie 02/12/16	2	Alberto Velázquez Alonso
5		Orquestación	33 días	mié 16/11/16	vie 30/12/16	3	Jonathan Sánchez Paredes
6		Pruebas preliminares con código	20 días	lun 05/12/16	vie 30/12/16	4	Alberto Velázquez Alonso
7		**Fase 2 - Desarrollo**	**86 días**	**lun 02/01/17**	**dom 30/04/17**		
8		Test de rendimiento Docker	35 días	lun 02/01/17	vie 17/02/17		Alberto Velázquez Alonso
9		Imágenes Docker	35 días	lun 02/01/17	vie 17/02/17		Jonathan Sánchez Paredes
10		Instalación KBN + Modificaciones	50 días	lun 20/02/17	vie 28/04/17	9	Jonathan Sánchez Paredes
11		Código: Diversas opciones	50 días	lun 20/02/17	vie 28/04/17	8	Alberto Velázquez Alonso
12		**Fase 3 - Documentación**	**26 días**	**lun 01/05/17**	**lun 05/06/17**		
13		Memoria	26 días	lun 01/05/17	lun 05/06/17		Alberto Velázquez Alonso[50%];Jonathan Sánchez Paredes[50%]

1. Introduction

Computation, since its inception, has witnessed many changes. From big mainframes that pioneered the field, with high costs in money and resources, to current personal computers that nowadays can surpass these old machines in performance. In contrast, personal computers have been gaining prominence over time and nowadays they can surpass the large mainframes. This evolution gave birth to distributed systems, where many small computers work concurrently and communicate to achieve a common task. Also, apply modularity to the application by separating the interface from execution.

Our school, 'Facultad de Informática', wants to be part of said evolution, by developing the 'Juez' application from a single host to a distributed system, with the purpose of enhancing the user experience and optimizing the servers' workload.

Online judges are used mainly for teaching activities. This software helps teachers assign homework to their students, so they write code pertaining to these assignments and sent it for grading. The judge compiles the code and presents the student with the result of said compilation, and at the same time stores the grades. Some virtual judge software alternatives are FLOP, DOMJudge, Mooshak, and AceptaElReto.

1.1. Motivation

During the last years, there have been advances on systems architecture towards distribution with the aim of sharing the load on servers, or even using and paying for just the needed resources, and not the whole machine. As the virtual judge, FLOP, is currently running on a single host, it is going to be ported to a distributed system.

A distributed system is a group of computers which are physically isolated but connected through a network. The user perceives this layout as a single system from the outside.

With the intent of easing distribution and orchestration of applications, Docker is used on the distributed system, and for its orchestration several software alternatives will be employed, such as Kubernetes, Docker Swarm or Mesos.

The main idea of this project is developing an execution environment that can be used by any virtual judge, including FLOP.

1.2. Goals

The main goal of this project is sharing the load of running many tasks between several nodes and achieve a higher efficiency, therefore serving a larger number of

requests. By being composed of several computers, the cluster does not need high per-machine performance, thereby reducing the cost. Another benefit is having a scalable and highly available system.

Under some circumstances we may need to execute unknown and untrusted code, so controlled environments such as virtual machines or software containers are frequently used. When the number of tasks is large, it is paramount to work with a distributed system to manage the different hosts and balance the load, to achieve maximum efficiency. The goal of this project is to develop a distributed system for the execution of untrusted code, providing an easy interface for its usage.

1.3. State of the Art

Searching for software with similar functionality, we found several alternatives, such as http://fiddles.io , http://ideone.com and http://codepad.org.

While these applications offer a similar service, there is no way to know their internals; they could work as a distributed system or operate with just a single host. The state of these alternatives is unknown.

It can readily be seen by searching online that the use of distributed, orchestrated systems is soaring right now in number of users, quickly becoming a mainstream technology.

VIRTUALIZATION

Virtualization involves the simulation of one computer within another by software, optionally with the help of special hardware devices. The virtual machine is displayed as a real machine in front of the programs it runs.

CONTAINER

A container is an isolated environment in which the processes that are executed are separated from the rest of the processes of the machine. The file system visible to the container processes is also independent.

DOCKER

Docker is currently the most popular container technology. Docker depends on several Linux features, and although it can be made to run on other operating systems, it does so under a virtual machine.

RKT

RKT is another container technology developed by the team at CoreOS, developed to work under their own operating system, and aimed towards security.

ORCHESTRATION

Orchestration is about automated container management, including launching, stopping and restarting. These containers can be hosted on a single machine or on several, depending on the features of the orchestration software being used.

KUBERNETES

Kubernetes is a container orchestration system developed by Google and designed to work with both Docker and RKT.

DOCKER SWARM

Docker Swarm is an orchestration system made by Docker, so therefore it only works alongside Docker containers. Its aim is to use the same API as Docker to automate the management of multiple containers.

APACHE MESOS

Mesos is a software whose goal is automating cluster management. It also can be used for container orchestration.

2. Requisitos y consideraciones previas

Para poner en contexto qué se va a desarrollar vamos a realizar unas consideraciones previas y los requisitos para el buen funcionamiento de esta.

- Arquitectura preexistente

La arquitectura con la que actualmente se ejecuta el código es un servidor web, en este caso Tomcat, que recibe un envío consistente en código fuente para ser evaluado.

Posteriormente el servidor envía de forma genérica el código a otra aplicación o servicio que ejecuta el código y devuelve el resultado de la ejecución, sea positivo o negativo el servidor almacena este resultado y se lo muestra al usuario que envió el código.

- Requisitos

Para el correcto funcionamiento de la aplicación tenemos que cumplir diversos requisitos tales como:

- Se lanza una tarea consistente en código fuente (Python en este caso, extensible a otros lenguajes).
- Se ejecuta en una máquina de un clúster.
- Se recupera el resultado para ser mostrado o almacenado (el almacenamiento está fuera del alcance del proyecto).
- La ejecución del código debe realizarse dentro de un contenedor Docker.

El contenedor debe reiniciarse o cerrarse después de la ejecución del código, de tal manera que no haya rastro de la ejecución anterior.

Debe controlarse el uso de recursos por parte del contenedor.

- Consideraciones sobre la arquitectura

Nomenclatura: “dispatcher” es el servidor Tomcat que acaba de recibir el código de un alumno para ser evaluado; “ejecutor” es cada instancia que espera a recibir código fuente para ejecutarlo y devolver el resultado.

Esquema de cola de tareas [5].

Para el inicio del sistema se realizaron unas primeras pruebas que consistieron en la recepción de la tarea, creación de nuevo contenedor con volumen compartido, ejecución y cierre, todo ello secuencial. Tras la primera prueba surgió un problema principal: penalización por cada una de las acciones, mucho tiempo entre el envío de la tarea y la recepción del resultado.

3. Tecnologías

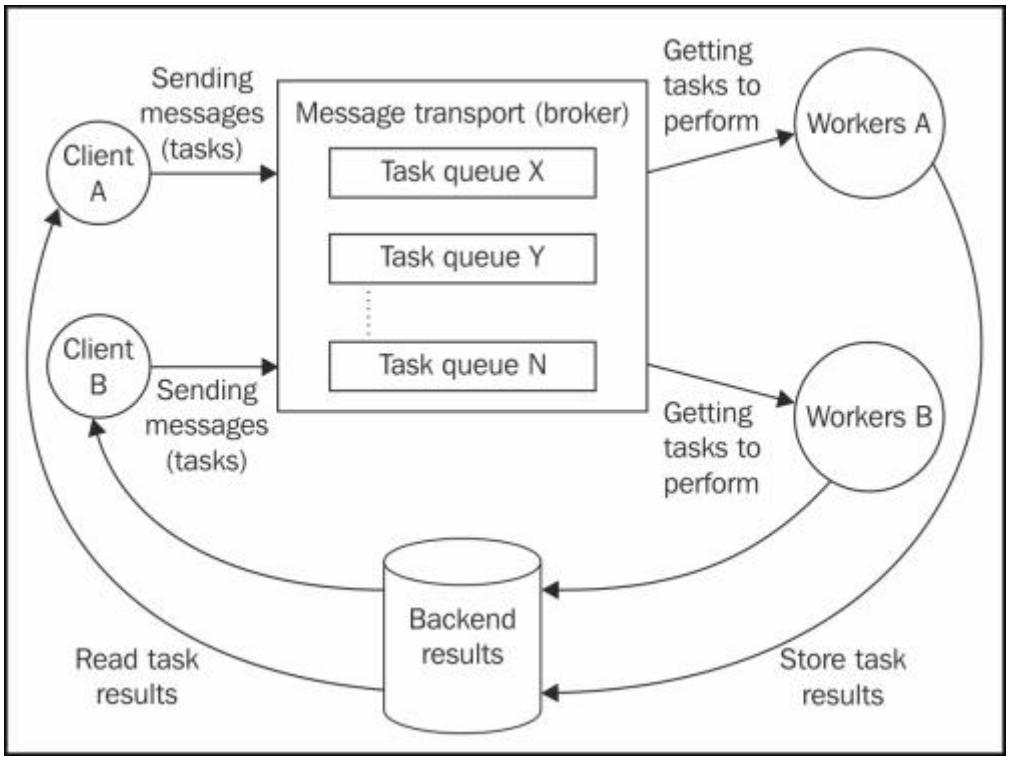
En este capítulo se detallan las tecnologías empleadas para el desarrollo del sistema distribuido.

3.1. Celery

Celery es una biblioteca de Python y un servicio asociado que implementan una cola de tareas.

Una cola de tareas es un tipo especial de cola que se utiliza para distribuir tareas entre diferentes procesos o hilos que se denominan "workers". Este tipo de cola presenta dos características relevantes que la diferencian de una cola genérica: la primera es que las tareas requieren confirmación por parte del sistema que gestiona la cola, de tal forma que si una tarea no se confirma se interpreta que el "worker" asociado ha sufrido un fallo y dicha tarea vuelve a la cola, quedando disponible para que otro "worker" la procese. La segunda característica es que se suelen utilizar como un tipo específico de sistema distribuido para enviar tareas a diferentes procesos o hilos que suelen ejecutarse en diferentes máquinas.

Celery está ideado desde un principio para ser ejecutado en diferentes nodos. Su modo de funcionamiento es bastante sencillo, al menos para el programador: sólo debe escribir una serie de funciones, que son las tareas que se ejecutarán en las diferentes máquinas, y lanzar el worker de Celery en cada uno de los nodos que vayan a ejecutar dichas tareas mediante el comando "`celery worker`". El programador, entonces, sólo necesitará llamar a dichas funciones con los parámetros relevantes para la tarea en cuestión; Celery distribuirá el trabajo entre todos los nodos disponibles y devolverá el resultado al programador.



Arquitectura general de Celery. Fuente: https://www.packtpub.com/mapt/book/application_development/9781783288397/7/ch07lvl1sec45/Understanding+Celery%27s+architecture

Caben distinguirse dos flujos dentro de cualquier cola de tareas y especialmente de Celery. El primero es el mecanismo de distribución de las tareas y de los parámetros con que se ejecutarán éstas, y el segundo es la recogida de los resultados. Celery no es una herramienta independiente, sino que para la implementación de estos flujos utiliza software ajeno, y de hecho permite utilizar diferentes tecnologías para cada uno de los dos mecanismos. La instalación recomendada utiliza *RabbitMQ* y *Redis*, si bien Celery soporta otras tecnologías como MongoDB, CouchDB en incluso bases de datos relacionales.

Para la distribución de mensajes a los "workers" se utiliza RabbitMQ. Más adelante se describe con más detalle esta tecnología, pero en esencia es una cola de mensajes [4], que permite la distribución de dichos mensajes desde varios productores hacia varios consumidores, y que permite implementar complejos modelos de distribución. En cuanto a los resultados, éstos se almacenan en Redis, que es un almacén de claves y valores.

La recomendación de estas tecnologías se basa en el hecho de que RabbitMQ es un software muy probado y estable, idóneo para la distribución de mensajes volátiles, mientras que Redis permite el almacenamiento y persistencia de las respuestas durante todo el tiempo que el cliente necesite hasta recuperarlas, y que además permite leer varias veces el mismo dato.

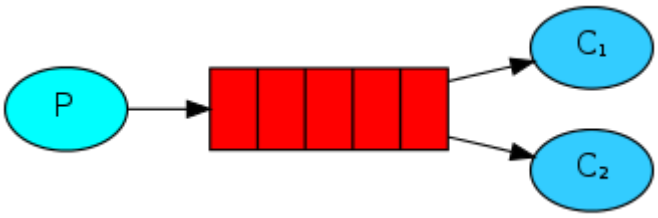
A nivel interno, el programador solicita la ejecución remota de una tarea, Celery le devuelve un objeto, que puede entenderse como un "futuro" o una "promesa". Una representación de una computación que puede o no haber terminado, y permite al cliente realizar otras tareas hasta que necesite el resultado de la ejecución de la tarea, ya que este modelo no bloquea el cliente. Una vez que el código cliente necesite el resultado puede solicitarlo explícitamente mediante una llamada bloqueante, que en este caso sí congela el hilo del cliente hasta que el resultado está disponible. Este modelo de programación permite por ejemplo lanzar varias tareas simultáneas para que sean ejecutadas en nodos diferentes y, una vez estén lanzadas todas, realizar otras operaciones independientes y después recuperar los resultados por orden.

3.2. RabbitMQ

RabbitMQ es una cola de mensajes. Una cola de mensajes es un mecanismo para el envío de datos entre hilos o procesos, a veces incluso entre procesos que se ejecutan en diferentes máquinas. Todas las colas de mensajes implementan alguna variante del patrón publisher/subscriber que podríamos traducir como publicador/suscriptor en el que uno o varios hilos productores generan datos que son enviados a uno o varios hilos consumidores.

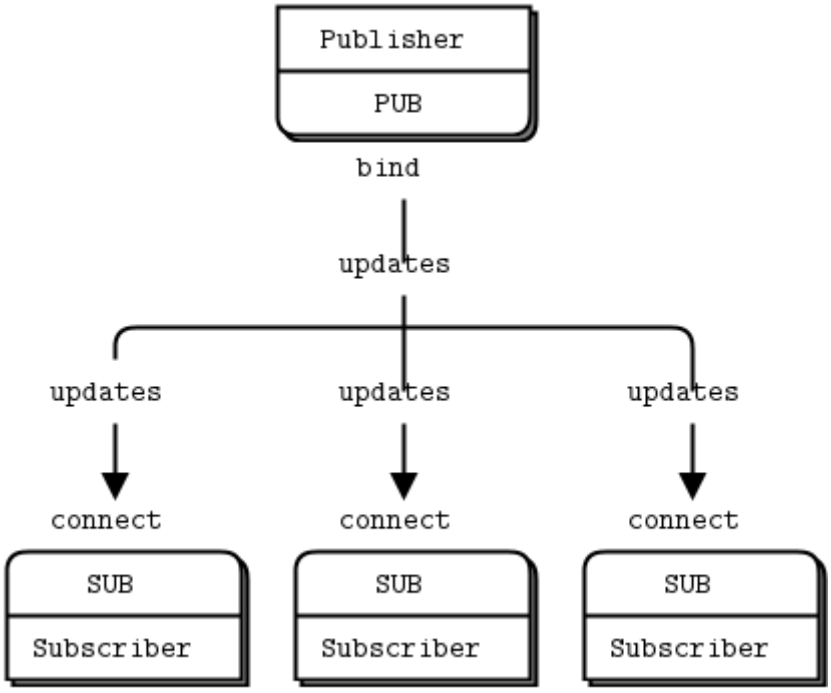
PUBLISHER / SUBSCRIBER

El patrón *publisher / subscriber*, también denominado pub/sub, define un escenario con productores y consumidores en el que los consumidores pueden especificar que no desean recibir todos los datos disponibles sino sólo aquellos que concuerdan con un patrón determinado.



Cola de mensajes pub/sub única. Fuente: https://www.rabbitmq.com/getstarted.html

RabbitMQ permite además implementar algunos patrones más avanzados de distribución de mensajes, como enrutamiento de mensajes, suscripción selectiva sólo a algunos temas o tipos de mensaje o RPC (petición/respuesta). RabbitMQ permite también algunas opciones avanzadas como la persistencia de mensajes, políticas de seguridad, notificación de recepción y la configuración explícita de diferentes parámetros como por ejemplo la política de entrega.



Patrón pub/sub detallado. Fuente: http://zguide.zeromq.org/page:all

Esta tecnología implementa un protocolo abierto denominado AMQP para la distribución de mensajes, lo que permite entre otras cosas que existan clientes para casi todos los lenguajes y plataformas en uso.

3.3. Redis

Redis es un servicio que almacena en tablas de hashes, conjuntos de clave y valor (*key-value store*) [9] en memoria RAM, el rendimiento puede ser muy elevado debido al almacenamiento en memoria RAM, comparándola con los motores de búsqueda de otras bases de datos.

IN-MEMORY DATABASE [8]

Una *in-memory database* almacena todos sus datos en memoria. Esto implica que las consultas son mucho más rápidas al no tener que leer ni escribir en un disco, pero como contrapartida el riesgo de pérdida de datos es mucho más elevado al no existir persistencia de datos.

Algunas bases de datos en memoria permiten el volcado periódico de los datos al disco, lo que reduce el impacto de un fallo inesperado a la pérdida de la nueva información generada desde el último volcado.

KEY-VALUE STORE

Al contrario que una base de datos relacional, una *key-value store* almacena pares clave-valor y no permite realizar búsquedas complejas, o al menos no tan complejas como SQL. Este tipo de bases de datos son idóneas para guardar datos sobre los que se sabe de antemano que no va a ser necesario realizar búsquedas complejas, ya que la velocidad de acceso e inserción es mucho mayor en comparación con las bases de datos tradicionales.

Específicamente, las consultas de tipo JOIN o similares son imposibles en este tipo de base de datos y suelen implicar consultar todos los datos y realizar la búsqueda programáticamente, lo que implica un mayor costo en memoria, eficiencia y esfuerzo del programador.

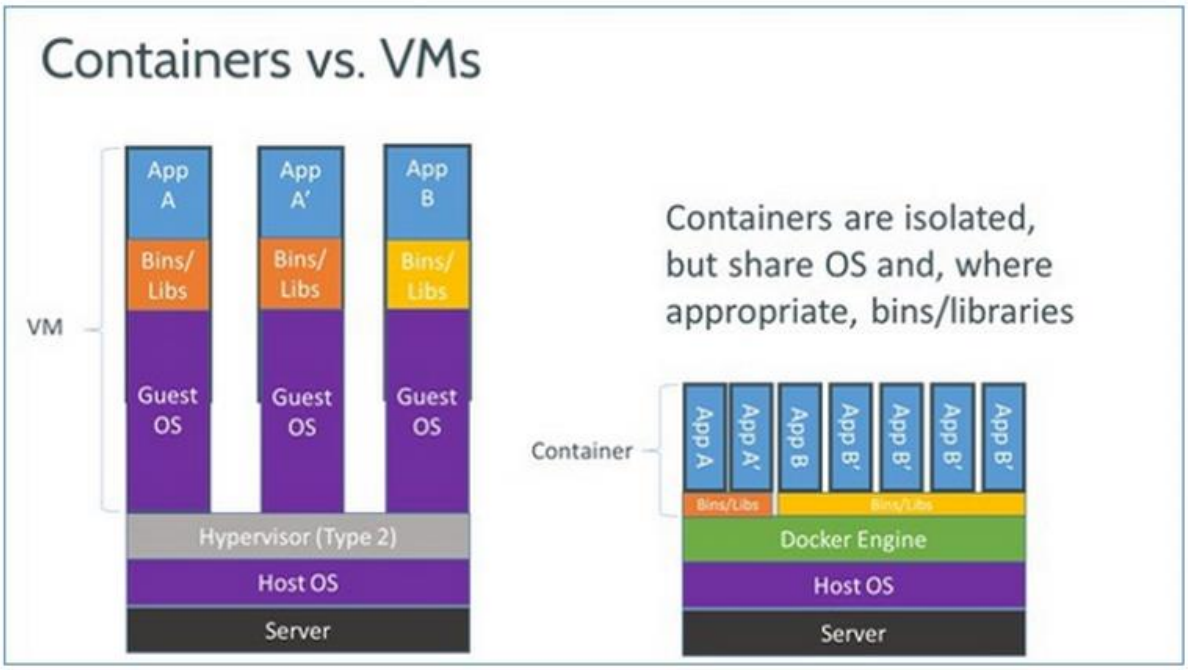
Esta tecnología no es una base de datos relacional porque no permite operaciones entre datos como JOINs ni consultas elaboradas mediante SQL. Sin embargo, ofrece el almacenamiento no sólo de cadenas de texto sino de diferentes tipos de datos abstractos como listas, conjuntos, tablas y otros tipos más especializados. Redis se ciñe estrictamente en establecer y recuperar datos sobre sus estructuras dejando de lado la búsqueda de relaciones o restricciones.

Redis permite una forma especializada de persistir los datos, así como replicación maestro-esclavo e incluso clustering con tolerancia a fallos de los nodos.

3.4. Docker

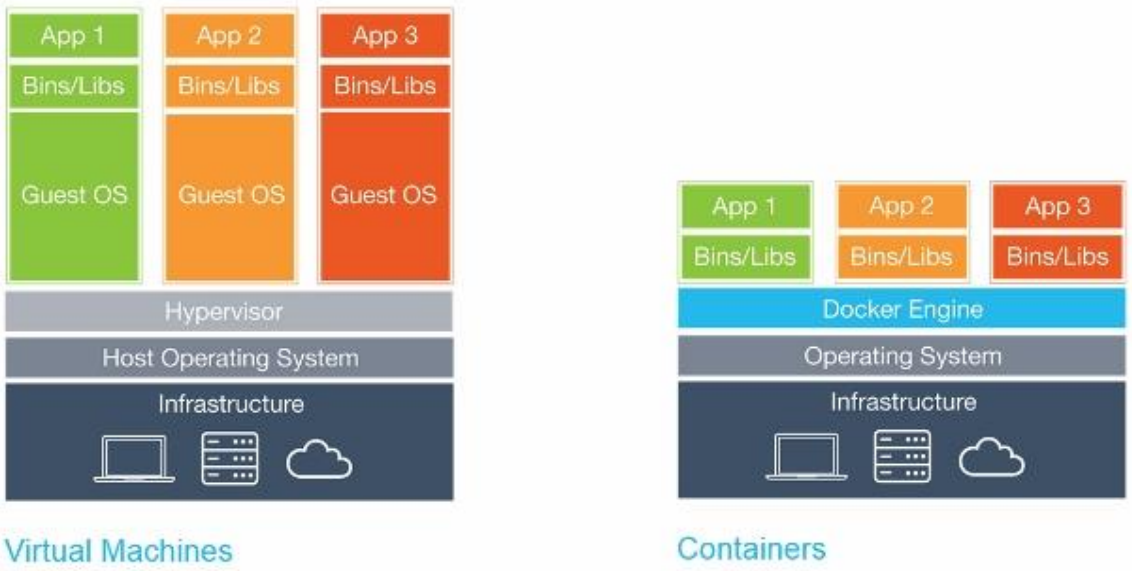
Docker es un software de virtualización y aislamiento de procesos a nivel de sistema operativo.

Al contrario que las máquinas virtuales tradicionales, Docker no emula completamente las llamadas al sistema y por tanto no necesita de instrucciones especiales para mantener un rendimiento aceptable. Los procesos que se ejecutan bajo Docker corren directamente en el sistema operativo *host* sin software intermedio, si bien el kernel se encarga de aislar dichos procesos para que no interfieran con el resto del sistema.



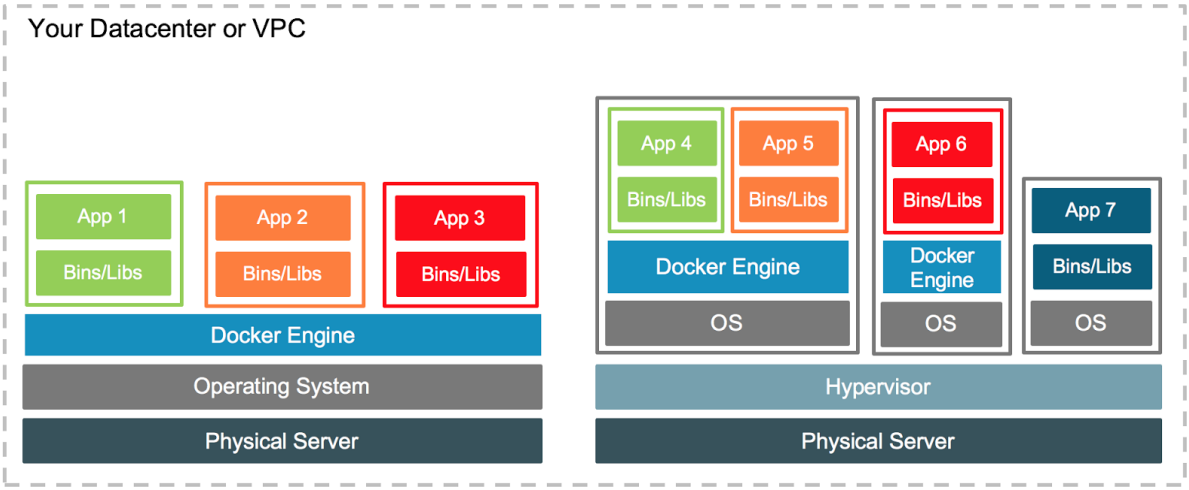
Diferencia entre máquinas virtuales tradicionales y Docker, como podemos ver docker funciona con el sistema operativo de la propia máquina y comparte los bins y librerías entre las aplicaciones del mismo tipo.

Fuente: http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/



Diferencia entre máquinas virtuales tradicionales y Docker. En esta imagen al ser aplicaciones diferentes, la única diferencia existente es que Docker funciona con el sistema operativo de la máquina, que lo lanza.

Fuente: https://www.docker.com/what-docker



Uso conjunto de Docker y máquinas virtuales. En esta imagen se ve claramente la ventaja de que Docker se lance con el sistema operativo de la máquina.

Fuente: https://blog.docker.com/2016/04/containers-and-vms-together/

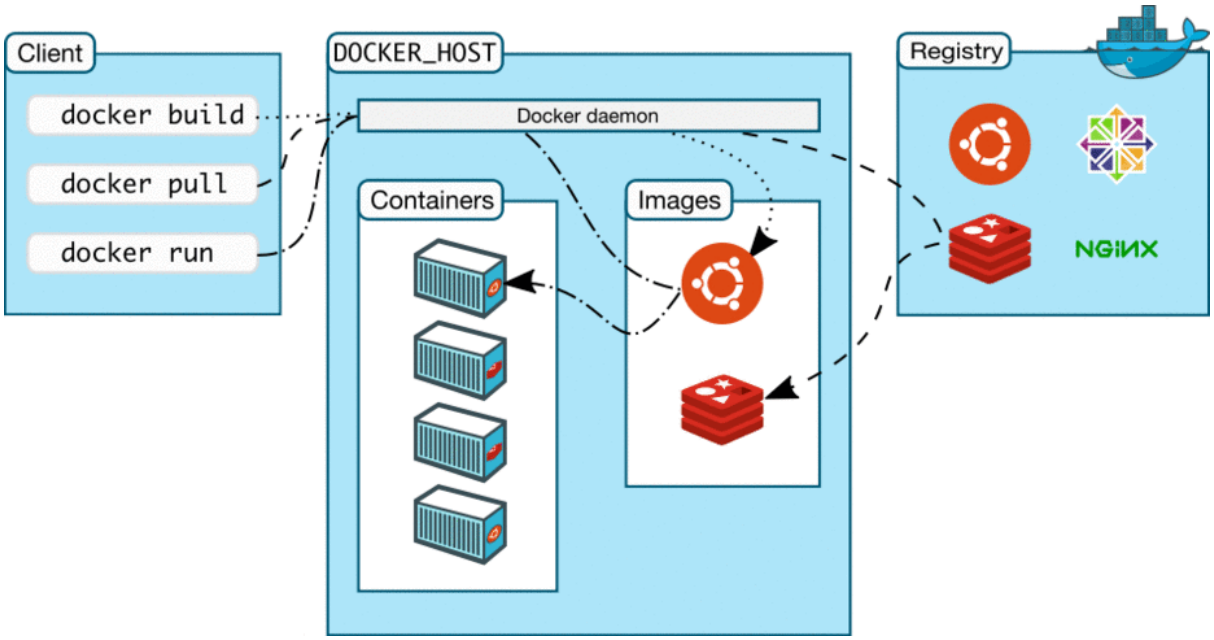
De momento Docker sólo funciona con el kernel de Linux, ya que éste proporciona tres características idóneas para aislar los procesos que corren dentro de los contenedores de Docker. Estas características son:

Los grupos de control ("cgroups") que limitan el acceso a recursos tales como memoria y CPU.

Los espacios de nombres ("namespaces"), que aíslan los procesos que corren dentro de los contenedores entre sí y de los procesos del host.

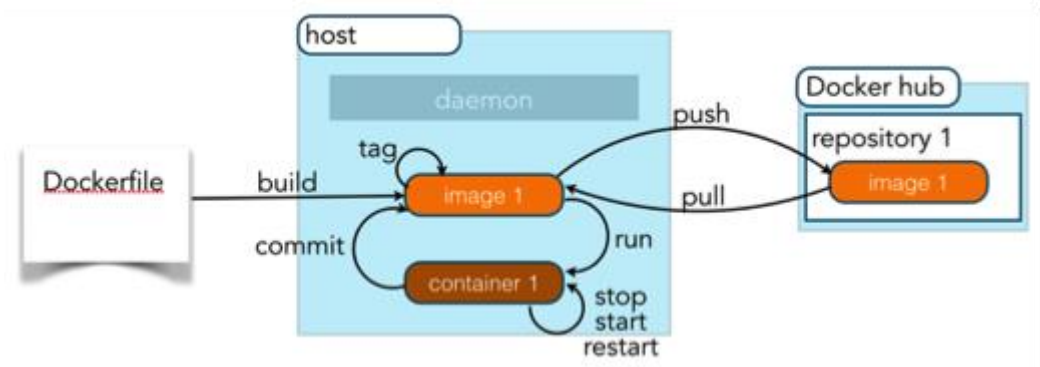
Un sistema de archivos como OverlayFS que no solo aísla los archivos que se encuentran dentro de un contenedor, sino que permite superponer los contenidos de múltiples sistemas de archivos, presentando un sólo sistema que resulta de la combinación de los contenidos de todos ellos.

Docker se gestiona a través de una API que puede ser llamada desde las herramientas de Docker en línea de comandos, desde bibliotecas creadas expresamente para diferentes lenguajes de programación, o bien realizando peticiones directamente al endpoint [2] del servicio web que proporciona dicha API.



Arquitectura de Docker. Fuente: https://docs.docker.com/engine/docker-overview/

Docker sólo puede lanzar imágenes creadas previamente. Existe un repositorio de imágenes creadas por la comunidad (Docker Hub) y otros repositorios mantenidos por diferentes entidades, por ejemplo, Google. Cualquier usuario puede generar sus propias imágenes y subirlas a Docker Hub simplemente creando un archivo llamado Dockerfile y ejecutando unos sencillos comandos que generan a partir de este archivo la imagen en la máquina local, que luego puede subirse a un repositorio. Para crear una imagen se necesita un archivo DockerFile. Este archivo contiene el sistema operativo y las herramientas que se indiquen. Para construir la imagen debemos hacer `docker build`. Una vez creado la imagen podemos subirla al repositorio haciendo `docker push`. Si la imagen está subida podemos descargarla con `docker pull`. Una vez tenemos la imagen se puede lanzar con `docker run` y realizar `commit` para guardar los cambios. La imagen siguiente explica estos pasos más gráficamente.



Fuente: https://dzone.com/refcardz/getting-started-with-docker-1

Hay que indicar Docker muestra un comportamiento que interesa destacar a la hora de lanzar o parar varios contenedores simultáneamente, como veremos en este

test de tiempos de arranque y parada de un contenedor en *dana*, el clúster de pruebas.

Comando de lanzamiento:

```
for i in $(seq 20); do
 /usr/bin/time -f '%e' docker run -dti --rm --name c$i debian >
/dev/null &
Done
```

Comando de parada:

```
for i in $(seq 20); do
 /usr/bin/time -f '%e' docker stop c$i > /dev/null &
done
```

run 1	run 2	run 3	run 4	run 5	stop 1	stop 2	stop 3	stop 4	stop 5
3.84	5.31	4.34	4.38	5.38	1.92	3.58	3.83	3.43	3.70
4.02	6.74	4.92	4.88	7.25	3.21	4.26	4.81	3.60	4.02
6.68	8.10	5.39	6.91	8.20	5.20	4.50	4.98	3.75	4.23
6.83	8.25	7.88	8.20	8.48	5.60	4.96	5.43	3.92	4.47
9.37	8.72	8.73	8.37	9.10	6.08	5.08	6.15	6.15	6.04
9.51	8.83	8.88	8.48	9.59	6.60	6.13	6.33	6.32	6.74
9.88	11.72	9.21	8.96	9.76	6.79	6.98	6.44	6.86	6.91
9.91	11.85	9.54	11.87	11.64	6.99	7.28	7.11	7.02	7.30
10.44	11.97	11.61	11.91	11.82	7.27	7.44	7.68	7.19	7.49
12.19	12.08	11.77	12.15	11.86	7.43	7.60	7.84	7.27	7.60
12.25	12.26	12.01	12.25	12.73	7.60	7.83	8.02	7.43	7.78
12.35	12.36	12.12	12.36	12.80	7.80	7.92	8.15	7.57	7.99
12.44	12.44	12.22	12.49	12.86	7.95	8.21	8.29	7.69	8.10
12.49	13.06	12.30	12.55	12.94	8.12	8.38	8.44	7.84	8.19
12.65	13.35	12.43	12.81	13.04	8.30	8.45	8.54	7.96	8.38
12.69	13.44	12.48	12.96	13.11	8.52	8.57	8.62	8.06	8.54
13.00	13.57	12.52	13.14	13.23	8.63	8.63	8.66	8.13	8.69
13.15	13.60	12.59	13.13	13.30	8.74	8.66	8.73	8.23	8.82

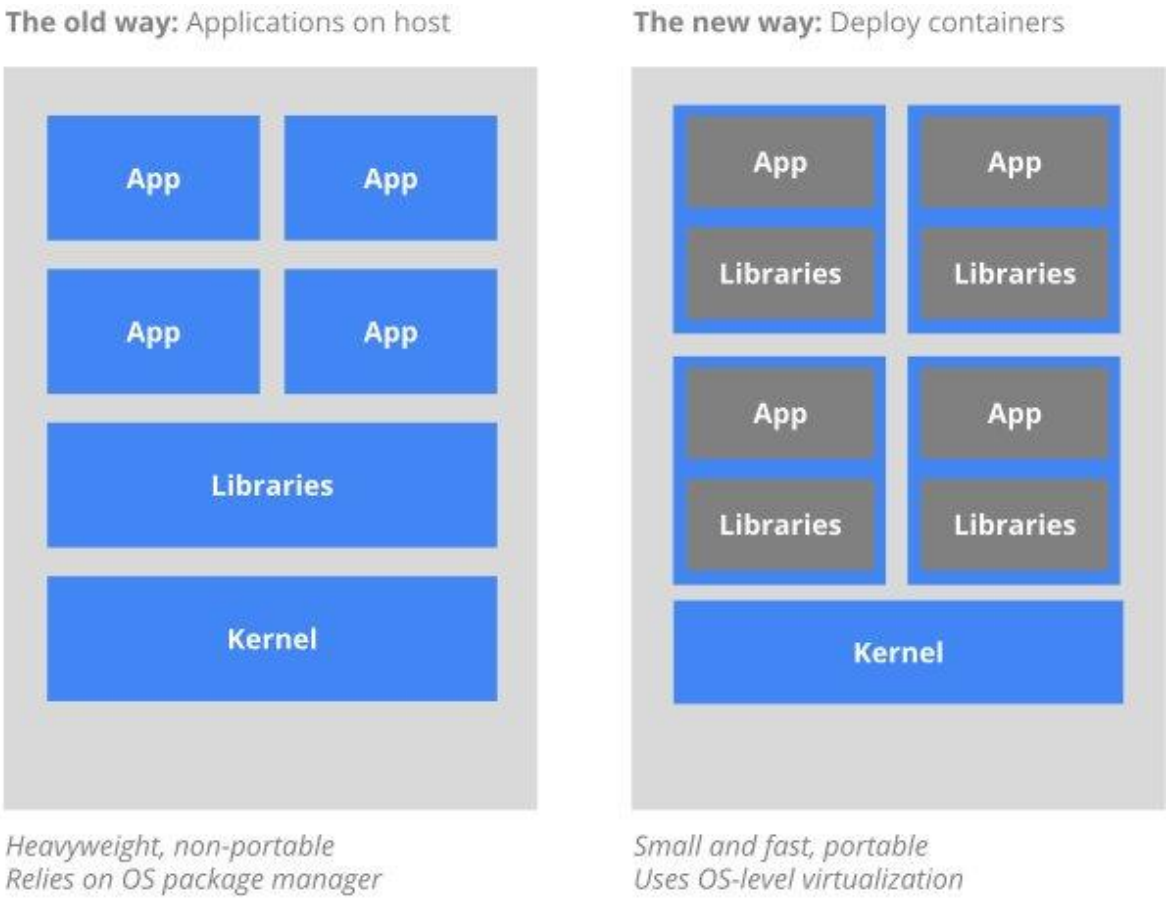
13.14 13.22	13.63 13.70	12.68 12.84	13.30 13.66	13.36 13.43	9.01 9.09	8.74 8.90	8.83 8.94	8.35 8.44	8.90 9.02

Hemos aprovechado un momento de elevada carga del sistema para realizar cinco rondas de lanzamiento y parada de 20 contenedores cada vez casi simultáneamente. Las cifras indican que, a medida que se acumula trabajo, Docker tarda cada vez más en arrancar y parar los contenedores. Este comportamiento es importante porque motivará más adelante algunas decisiones sobre la arquitectura de nuestra solución.

Es interesante mencionar también que Docker no es el único sistema de "containerización" existente; existen otras alternativas, como por ejemplo rkt. Rkt fue creado tras encontrar diversos fallos de seguridad en Docker, este nuevo sistema ofrece seguridad en las imágenes del contenedor, previene ataques de escalado de privilegios, más flexibilidad en la publicación de imágenes y portabilidad a otros sistemas de "containerización".

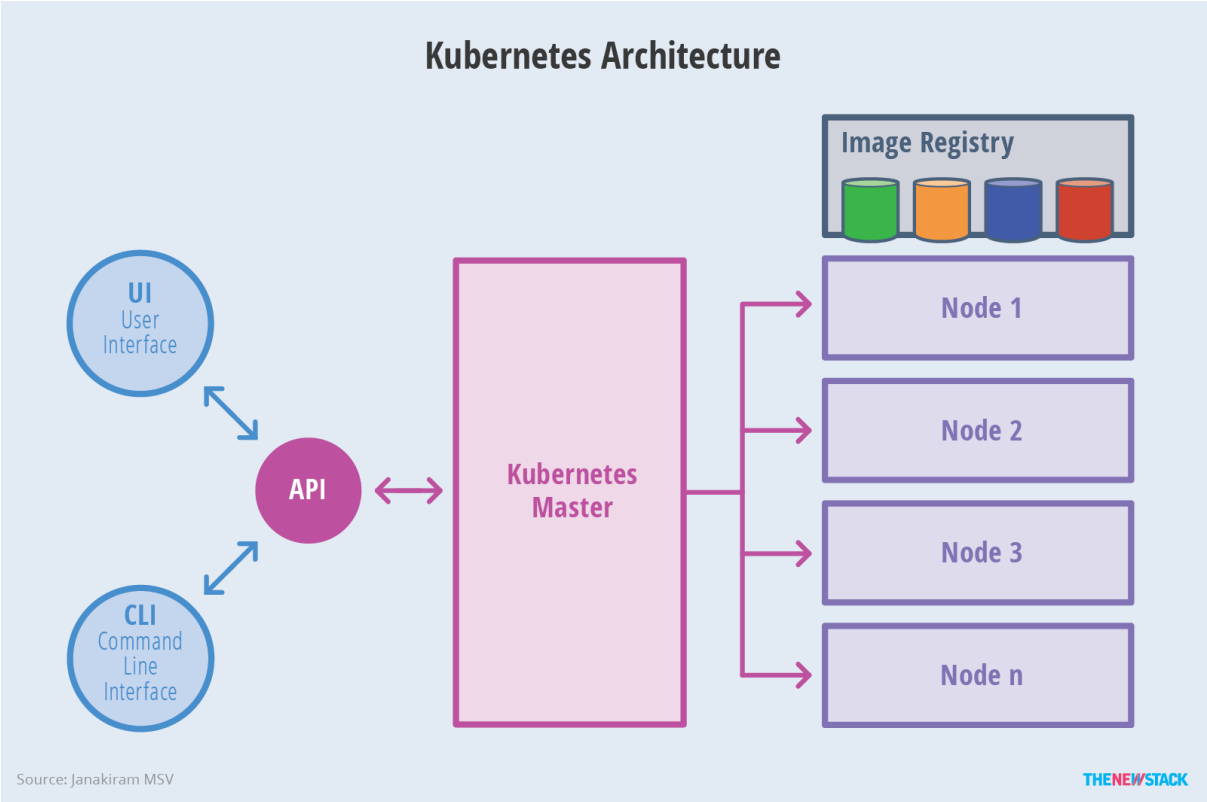
3.5. Kubernetes

Kubernetes, también llamado K8s [25], es un sistema de orquestación de aplicaciones que se ejecutan dentro de contenedores.

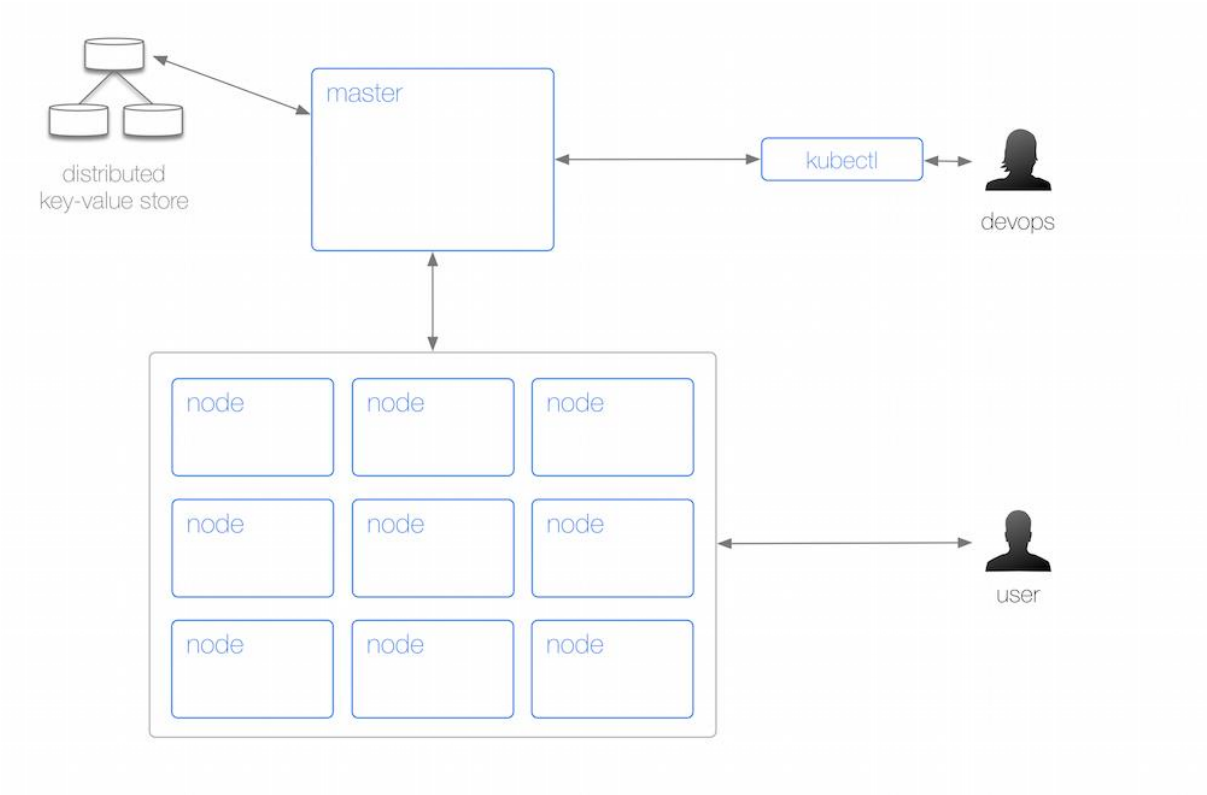


Qué aportan los contenedores. Como se puede ver en la figura tener contenedores es una gran ventaja, reducimos en módulos más pequeños, rápidos y portables. Fuente: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

La orquestación de contenedores consiste en la creación automática de contenedores, así como de su escalado, en función de la configuración especificada por el usuario, sobre múltiples nodos de un clúster o máquinas independientes.



Arquitectura general de K8s. Fuente: https://thenewstack.io/kubernetes-an-overview/



Interacción entre componentes de K8s. Fuente: http://k8s.info/cs.html

Kubernetes hace uso de varios conceptos únicos a esta tecnología que es necesario conocer para su uso efectivo. El más básico de ellos es el “pod”, que es uno o varios contenedores, como, por ejemplo, contenedores Docker. Alojados en la misma máquina física y que comparten recursos. Varios contenedores dentro de un pod comparten una dirección IP y un puerto, pueden conectarse a través de localhost, también pueden comunicarse mediante semáforos y memoria compartida. Las aplicaciones que son lanzadas dentro de un pod tienen acceso a los volúmenes compartidos, que han sido especialmente montados dentro del pod para este uso. Los pods no son destinados a ser tratados como entidades de larga duración, ya que no son capaces de resistir a fallos de programación, caídas de nodos, pero un usuario no tiene la necesidad de crear pods, normalmente crean controladores que son capaces de resucitar con un ámbito de clúster, así como la replicación y la gestión de despliegue.

Existe también el controlador o “controller”, que es una entidad abstracta encargada de crear nuevos contenedores, posiblemente en diferentes máquinas, según una configuración determinada. También es su tarea detectar si un contenedor se cerró y en caso necesario lanzar otro en su lugar, siendo capaz de tener un número de réplicas de pod constante, para garantizar que siempre estén disponibles. En conclusión, es el encargado de supervisar uno o varios pods a través de múltiples nodos, manteniendo siempre el sistema operativo para su uso. En el apartado de manual de instalación existe un ejemplo de “Replication controller”.

Un tercer concepto es el servicio o “service”. Los pods son mortales y cuando mueren no son resucitados, el controller crea y destruye pods constantemente, debido a esto nacen los servicios, que son una interfaz unificada para un pod o conjunto de pods, de tal manera que se presenten al exterior como un único servicio con una dirección de IP virtual y que internamente proporciona balanceo [14] de carga y DNS si es necesario.

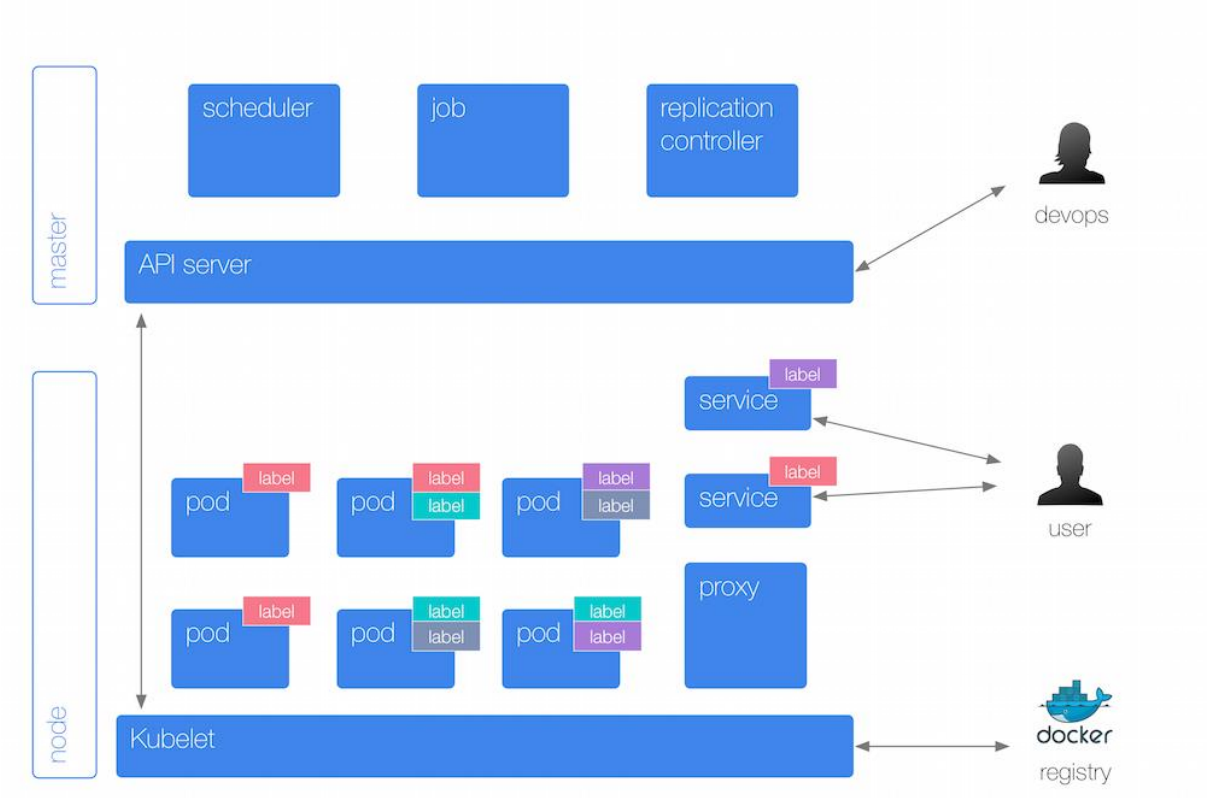
Para la creación de cualquiera de estos tres conceptos, simplemente tenemos que crear un archivo .yaml, que es un formato de serialización de datos, muy similar a XML o Python.

A continuación, un ejemplo para la creación de un pod, service o controller, donde podemos tratar todas las especificaciones de cada uno de ellos.

```
# Número de versión del api que se quiere utilizar
      apiVersion: v1
      # Tipo de fichero que se va a crear.
      kind: ReplicationController || kind: Pod || kind: Service
      # Datos propios del replication controller
      metadata:
            # Nombre del Replication Controller
            name: my-nginx
```

```
# La especificación del estado deseado que queremos que
tenga el pod.
spec:
   # Número de réplicas que queremos que se encargue de
mantener el rc. (Esto creará un pod)
     replicas: 1
     # En esta propiedad se indican todos los pods que se
va a encargar de gestionar este replication controller. En
este caso, se va a encargar de todos los que tengan el
valor "nginx" en el label "app"
     selector:
     app: nginx
     # Esta propipedad tiene exactamente el mismo esquema
interno que un pod, excepto que como está anidado no
necesita ni un "apiVersion" ni un "kind"
     template:
     metadata:
          name: nginx
          labels:
               app: nginx
     spec:
          containers:
               - name: nginx
               image: nginx
                ports:
                  - containerPort: 80
 #Aquí se define la polítca de restauración en caso de que
el pod se detenga o deje de ejecutarse debido a un fallo
interno
               restartPolicy: Always
```

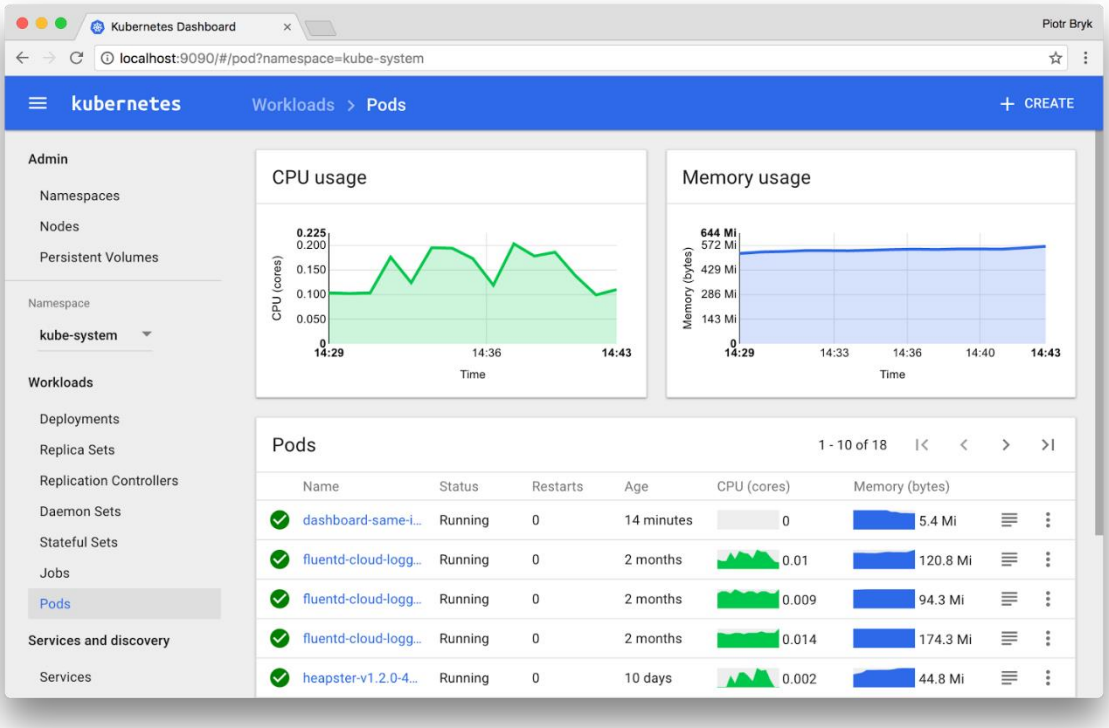
Kubernetes puede utilizarse también para gestionar otras tecnologías de contenedores, como rkt.



En la imagen podemos ver cómo los usuarios se conectan con los servicios, el administrador de DevOps con la API de Kubernetes, y Docker con Kubelet, dónde Docker puede ser sustituido por cualquier otro gestor. Arquitectura de Kubernetes desde el punto de vista de los diferentes perfiles de usuarios.

Fuente: http://k8s.info/cs.html

Para una versión más gráfica podemos incorporar Web UI que es un Dashboard que nos proporciona Kubernetes, se puede reiniciar un pod, crear o modificar recursos de Kubernetes, implementar nuevas aplicaciones, información sobre el estado de recursos. En conclusión, se puede realizar cualquier operación sobre nuestro sistema de una forma más gráfica y sencilla.



Fuente: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

4. Decisiones arquitectónicas

En este apartado se realizará una investigación entre las tecnologías más punteras que existen en la actualidad, con el fin de contestar las siguientes preguntas:

- Que orquestación tendrá el sistema distribuido.
- Tipo de cliente.
- Localización del cliente.

4.1. Orquestación

ORQUESTACIÓN

En el ámbito de los contenedores software, se entiende como *orquestación* a la gestión unificada y automatizada de dichos contenedores. El software de orquestación recibe una configuración deseada y se encarga de llevarla a cabo en los nodos que gestiona, levantando contenedores de imágenes específicas cuando es necesario y reiniciándolos si es preciso.

Los sistemas de orquestación proporcionan funcionalidades adicionales a la mera gestión de contenedores; por ejemplo, permiten exponer grupos de contenedores como un único servicio, así como balancear la carga entre ellos.

Para la elección de qué tecnología se utilizará para orquestar el sistema distribuido se realizará una investigación exhaustiva de las tecnologías del mercado actual.

Para una primera toma de contacto se realizará una breve descripción, basada en las funcionalidades de las tecnologías más punteras.

Las siguientes imágenes han sido tomadas de las diapositivas de la página:

https://es.slideshare.net/KarlIsenberg/container-orchestration-wars

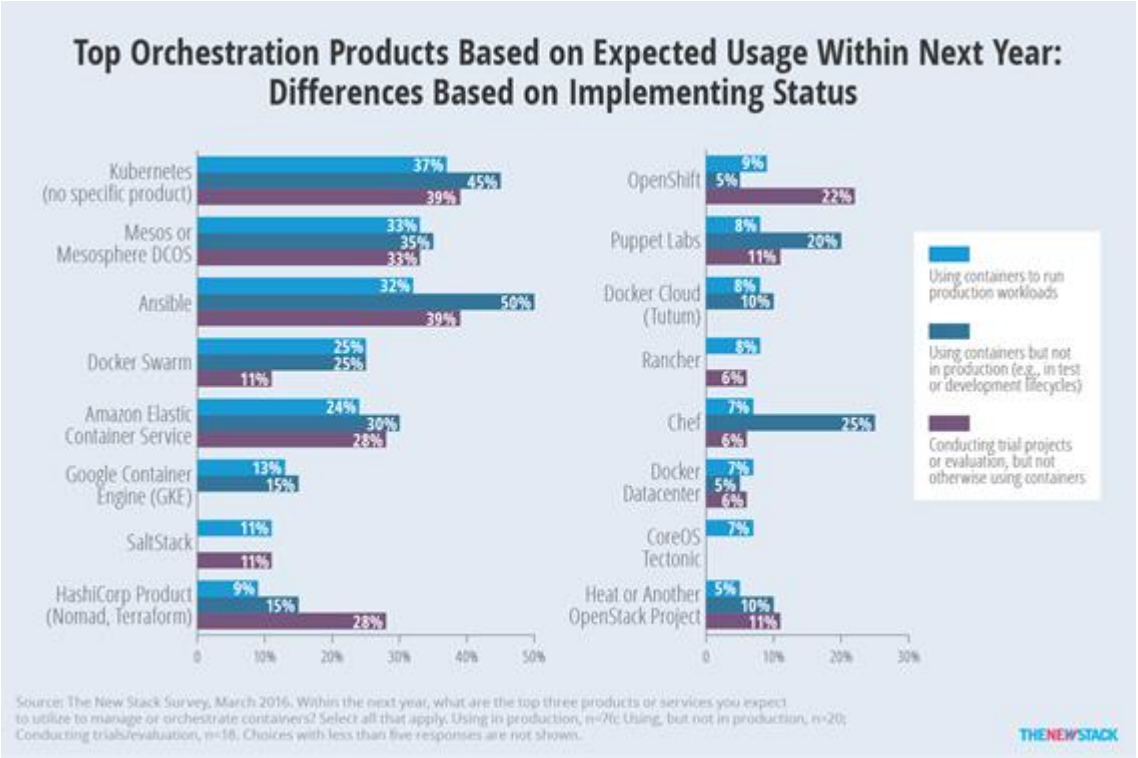
El primer punto que se va a analizar es la programación, la capacidad de cada tecnología en la escalabilidad, resurrección, despliegue continuo y colocación de los nodos.

Como se puede observar en este aspecto la tecnología más completa es Kubernetes, ya que tiene todos los puntos incluyendo la resurrección de nodos que es uno de los requisitos del sistema, otras dos tecnologías con bastantes puntos a favor, son Swarm y Mesos/Marathon. El siguiente punto a analizar es la administración de servicios, la cual para nuestro sistema distribuido es muy importante, ya que en los requisitos se pidió que se controlarán recursos como el consumo de CPU, limitar el espacio de disco o el tiempo de ejecución de una máquina.

En esta imagen podemos ver cómo la tecnología que más se amolda a nuestras peticiones es Mesos con Marathon y justo detrás de ella Kubernetes y Swarm de nuevo, por lo tanto, estas tres tecnologías cogen muchos puntos para ser elegidas. Por último, vamos a analizar la administración de servicios, como, por ejemplo, la carga balanceada y las etiquetas.

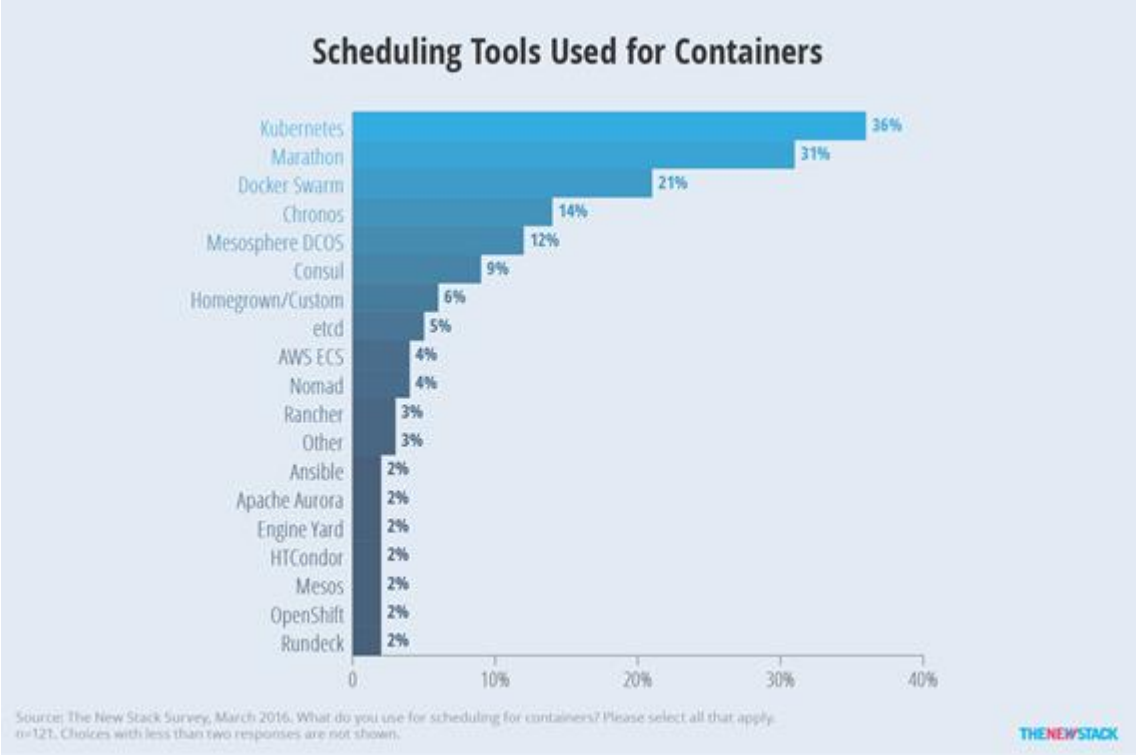
De nuevo las tecnologías que más puntos a su favor tienen son Kubernetes y Mesos con Marathon.

Para completar la investigación veremos dos gráficas más sobre las tecnologías más usadas en el mercado actualmente y a continuación describiremos una a una cada tecnología. La primera de ellas son las tecnologías que orquestan, en primer lugar, si nos fijamos en la línea de azul claro, se observa que las más utilizadas en producción son Kubernetes y Mesos. En cambio, si nos fijamos en la línea de azul oscura, se observa que los porcentajes son mayores en no producción, dando a entender que es una tecnología en crecimiento, todavía está en pruebas.



Fuente: https://thenewstack.io/tns-research-present-state-container-orchestration/

En esta segunda imagen podemos observar las herramientas de programación, compañeras de las de orquestación, en primer lugar, se encuentra Kubernetes y el compañero de Mesos, Marathon. También, destacar Swarm que es distribuido por Docker.



Fuente: https://thenewstack.io/tns-research-present-state-container-orchestration/

- **Docker Swarm**

Es la solución más simple, viene instalada con Docker Engine, ya que es una distribución oficial de docker, a partir de la versión 1.12. Es compatible con la API de Docker, pero se trata de una tecnología muy poco madura para entornos de producción.

- **Mesos con Marathon**

Se trata de una mezcla de tecnologías donde Marathon es el encargado de la orquestación del sistema y Mesos, en cambio, lanza los contenedores de imágenes de Docker, también podemos intercambiar Mesos por Kubernetes, creando una tecnología potente pero pesada. Se trata de una tecnología muy madura, al respecto de las demás. Pero tiene un consumo muy elevado de recursos.

- **Yarn**

Yarn es otro gestor de sistemas distribuidos, pero fue creado con la idea de utilizarse para Hadoop.

- Nomad

Nomad es la tecnología más reciente de todas, con menos de un año. Es la más simple tiene un binario que puede actuar como cliente y servidor, pero al ser tan joven apenas hay documentación, muy poca difusión y asume que se va a utilizar con Vagrant.

4.2. Tipo de cliente

En este apartado se toma la decisión del tipo de cliente que se va a emplear para el desarrollo del sistema distribuido y las diferentes posibilidades existentes.

4.2.1. Servicio Web

Una alternativa para enviar programas a los nodos que los van a ejecutar y posteriormente recuperar los resultados es mediante micro-servicios [1], utilizando la arquitectura REST. Este sistema es relativamente sencillo de implementar ya que en casi cualquier lenguaje existen bibliotecas que permiten la creación de un servicio web y la conexión al mismo, y además el protocolo utilizado, HTTP o HTTPS, es público, abierto y extremadamente estable, lo que permitiría la interoperabilidad de forma muy sencilla en caso de que se quisiera extender o modificar el programa en un futuro.

MICROSERVICIOS Y REST

La arquitectura de microservicios es un patrón por el cual una aplicación se divide en módulos o fragmentos que se comunican entre sí mediante peticiones HTTP. La ventaja principal es que obliga a desarrollar sistemas con bajo acoplamiento, a costa de añadir un cierto retraso por las peticiones HTTP y la serialización de los mensajes.

El protocolo HTTP se desarrolló para la entrega de hipertexto y no para la comunicación entre módulos o procesos, por lo que se desarrolló un marco conceptual para la adaptación de HTTP a la comunicación entre módulos. Este marco se llamó *REST*, *representational state transfer*, y establece una correlación entre comandos HTTP y operaciones entre módulos. En REST, las peticiones GET, POST, PUT y DELETE equivalen a los conceptos crear, leer, actualizar y borrar.

En este punto se nos plantean dos alternativas: podemos tener un servidor en el nodo maestro y clientes en los nodos ejecutores, o al revés. Estas alternativas tienen cada una sus pros y contras, que pasaremos a explorar.

Tal vez la alternativa más inmediata sea tener un micro-servicio en el nodo que recibe las peticiones de los usuarios y que los nodos que ejecutan las tareas soliciten los datos a dicho servicio (GET) y posteriormente devuelvan los resultados al mismo (POST). El problema principal de esta arquitectura es que, aunque el servidor no tenga tareas que enviar los clientes deben consultar periódicamente si hay nuevas tareas que ejecutar, creando una mínima carga residual en el sistema.

Otro problema derivado del anterior es el hecho de que lleguen múltiples tareas mientras todos los clientes están dormidos, en cuyo caso dichas tareas no comenzarán a ejecutarse hasta que llegue el momento de que los clientes vuelvan a solicitar tareas al servidor, por lo que dependiendo la configuración de los delays puede tardarse un tiempo considerable en servir una petición de un usuario. Creemos que este fenómeno se daría con cierta frecuencia según los patrones típicos de uso de esta aplicación: los alumnos tienden a enviar en masa las tareas para ser evaluadas el último día e incluso la última hora. Hay que decir que este problema no se presentaría con numerosas solicitudes de usuarios en espera, sino sólo con un flujo bajo pero constante de tareas pendientes.

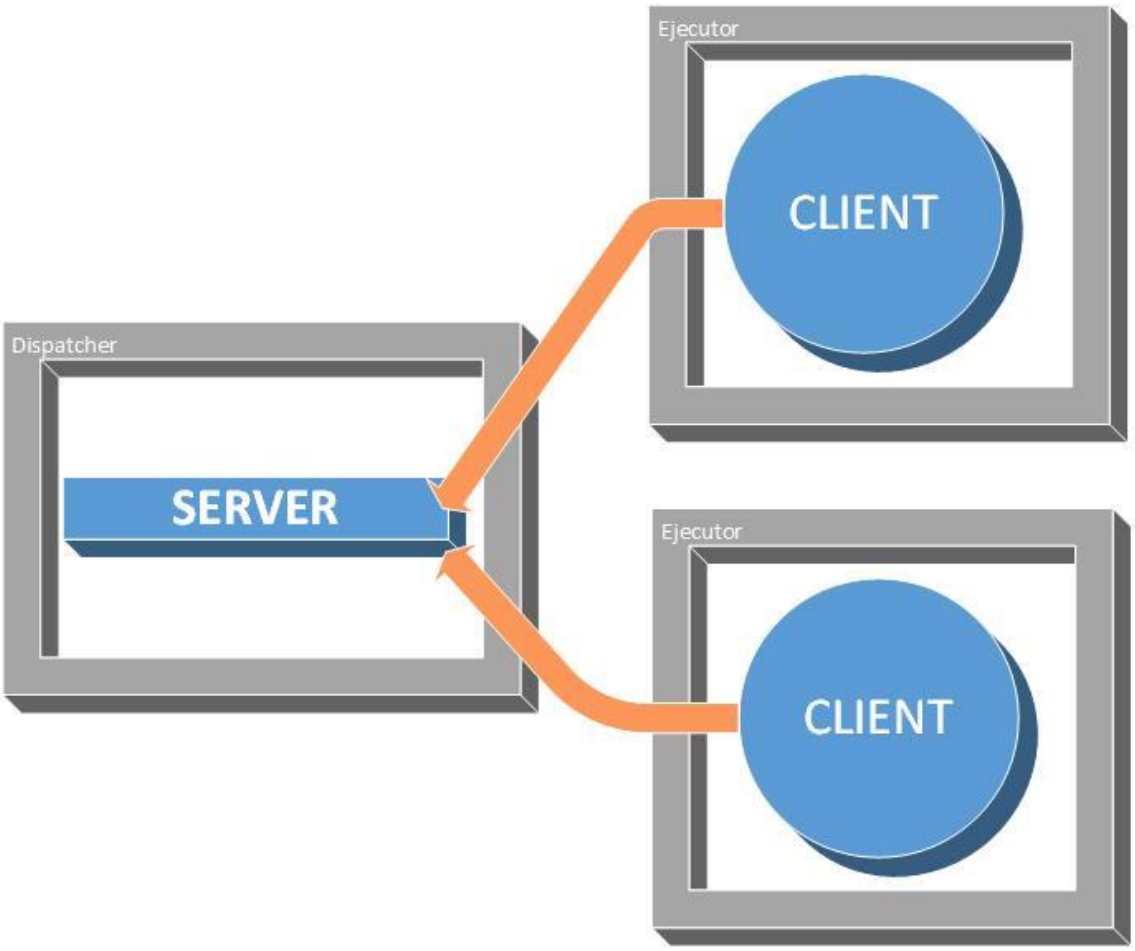
ESPERA ACTIVA Y PUSH / PULL [6]

En un escenario en el que hay productores y consumidores, el evento en el que un consumidor solicita datos a un productor se denomina *pull*, mientras que, si es el productor el que inicia la comunicación de los datos disponibles a los consumidores o les señaliza dicho evento de alguna forma, se denomina *push*. Cada una de las dos variantes tiene diferentes características de rendimiento.

Cuando ningún productor tiene datos disponibles y tampoco dispone de ningún método de señalizar a los consumidores la presencia de nuevos datos, los clientes suelen tener que realizar *espera activa*, también denominada *polling* [21]. Ésta consiste en que los consumidores solicitan datos a los productores en bucle, lanzando tantas peticiones como el sistema permite por lo que el rendimiento del sistema se degrada. Una alternativa es esperar un tiempo determinado hasta lanzar la siguiente petición, lo que hace que el rendimiento del sistema disminuya.

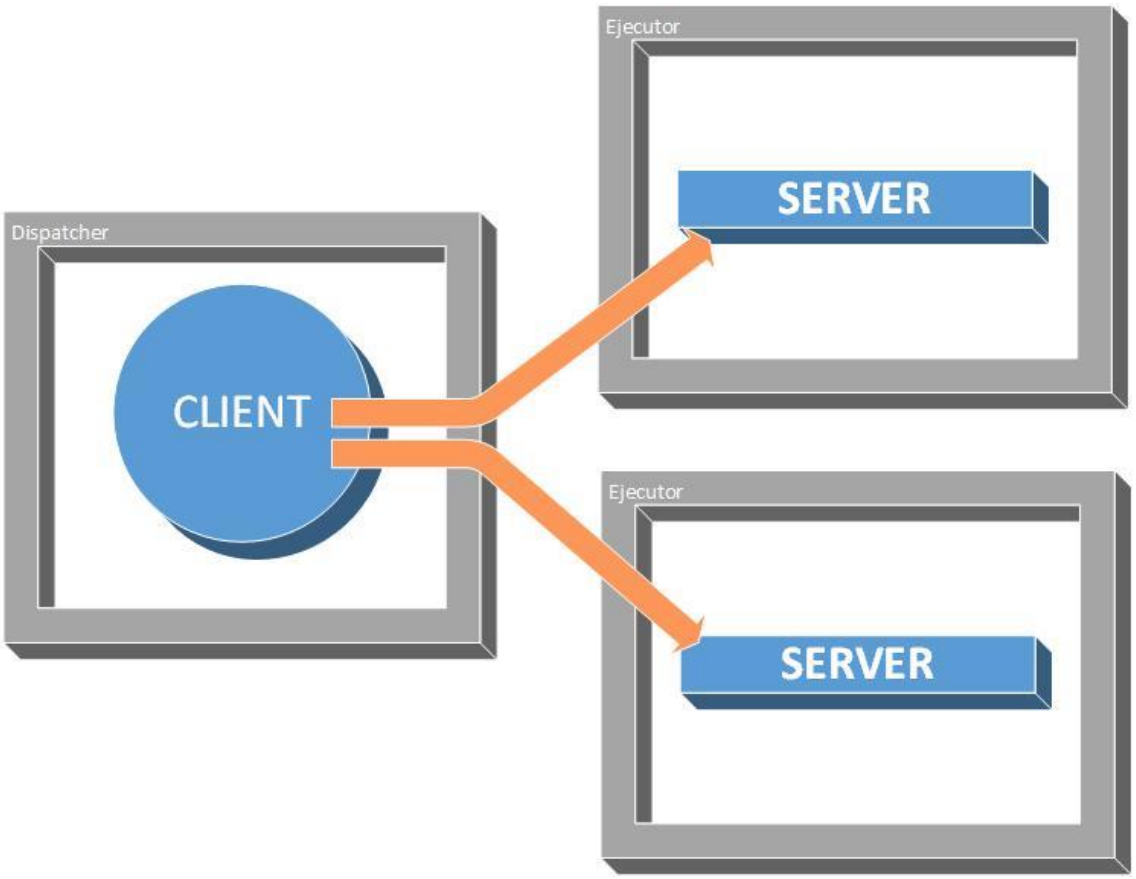
En contraste, si existe algún método para señalizar la presencia de datos, los consumidores pueden bloquear y mientras se encuentran bloqueados no consumen recursos. Los métodos de espera más comunes son variables de condición si sólo se necesita espera entre hilos, o bien las llamadas del sistema *select* o *epoll* si se necesita sincronización entre procesos, e incluso entre diferentes hosts.

En la imagen podemos ver el modelo gráfico de un micro-servicio, donde el servidor está en el nodo *host* y los nodos ejecutores son los clientes.



La otra alternativa utilizando una arquitectura REST [3] sería a la inversa: el nodo que recibe las solicitudes de los usuarios sería el cliente, y los nodos que ejecutan las tareas serían los servidores. Esta arquitectura aparentemente extraña tendría la enorme ventaja de no depender del delay del cliente, ya que las tareas serían enviadas a los clientes disponibles a medida que fueran llegando sin depender de retrasos artificiales, pero tendría varias desventajas. La primera, el nodo "dispatcher" debería mantener una lista de nodos ejecutores disponibles, por ejemplo, contando con un micro-servicio en el que se registrara cada nodo al arrancar y/o al quedar disponible después de haber ejecutado una tarea. Esto complicaría la implementación y obligaría al programador a implementar un mecanismo para llevar el registro de nodos disponibles y tareas pendientes, así como controlar aquellas tareas enviadas, pero no finalizadas.

En la imagen podemos ver el modelo gráfico de un micro-servicio, donde el servidor está en los nodos ejecutores y el nodo *host* es el cliente.



4.2.2. Websockets

WEBSOCKET [23]

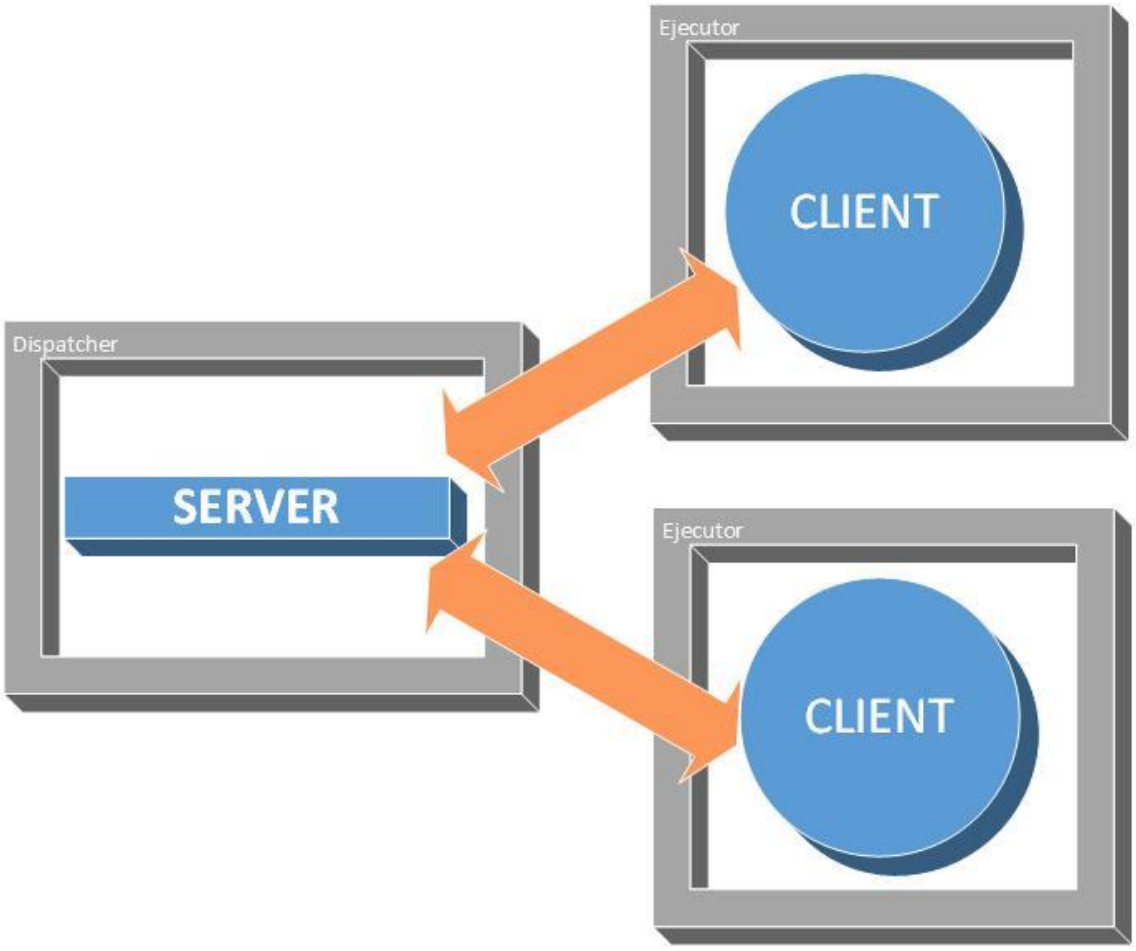
Websocket es una tecnología de comunicación bidireccional desarrollada primeramente para navegadores web, pero que en realidad es independiente de HTTP. El protocolo de Websocket establece una conexión permanente entre cliente y servidor que permanece abierta durante toda la sesión, lo que implica un mejor rendimiento, pero también un mayor consumo de recursos en el lado del servidor.

Los motivos para el desarrollo de este protocolo son principalmente dos: por un lado, permitir al servidor enviar datos al cliente sin petición previa, y por el otro lado evitar que el cliente tenga que establecer una nueva conexión cada vez que desee realizar una petición al servidor.

Una modificación de la implementación mediante micro-servicios es la utilización de *websockets*. Aunque no es un protocolo tan probado como HTTP

también dispone de bibliotecas en muchos lenguajes y permite comunicación bidireccional entre dos puntos, así como bloquear el cliente a la espera de datos, lo que alivia las desventajas de las dos alternativas anteriores. Sin embargo, el programador sigue siendo el responsable de controlar las tareas pendientes y el estado de cada nodo. En este punto no hay una ventaja clara en cuanto a usar *websockets* frente a las alternativas que veremos más adelante.

En la imagen podemos ver el modelo gráfico de un micro-servicio, donde la conexión entre el nodo ejecutor y *host* es bidireccional.



4.2.3. Celery

COLA DE MENSAJES Y COLA DE TAREAS

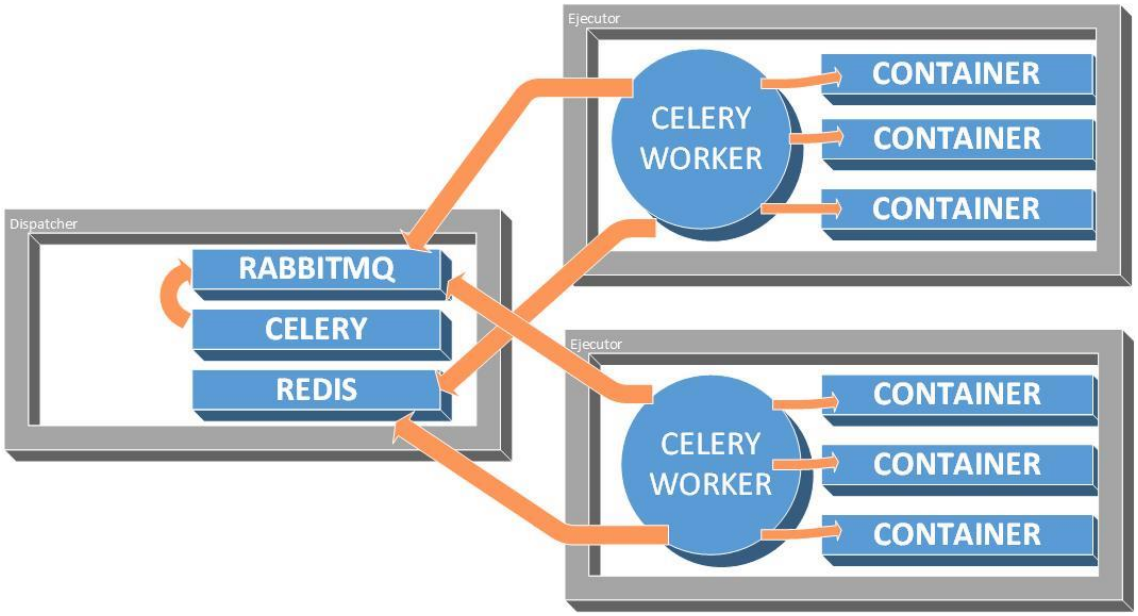
Genéricamente, una cola es una estructura de datos que permite insertar datos por un extremo y extraerlos por el otro. Es por tanto una estructura FIFO.

Una cola de mensajes tiene características adicionales a una cola normal. Su función es la distribución de mensajes entre procesos, y permite que los clientes se bloqueen cuando la cola está vacía. Tiene otras características interesantes como permitir que los clientes se suscriban sólo a ciertos mensajes y no a todos (ver publisher / subscriber).

Una cola de tareas es, al menos conceptualmente, una extensión de la cola de mensajes que además requiere que, al recibirse una tarea, los clientes lo notifiquen.

Celery es un servicio que proporciona una cola de tareas, desarrollado específicamente para ser utilizado desde Python. Internamente utiliza RabbitMQ y Redis.

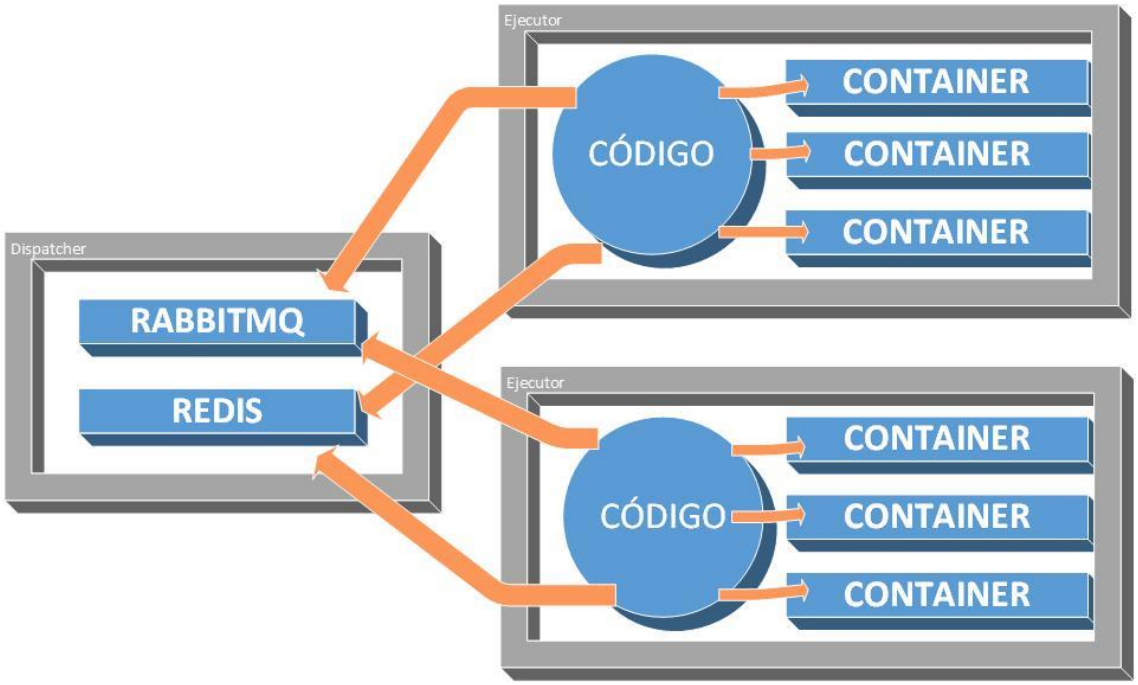
Las ventajas principales de este sistema son múltiples: por un lado, permite a los clientes esperar hasta que haya datos que procesar; provee de un mecanismo muy simple de balanceo de carga mediante *prefetching*, y además incorpora un sistema implícito de notificaciones, de tal manera que una tarea enviada a un worker que falla será reenviada a otro worker al no recibir confirmación.



4.2.4. RabbitMQ con Redis

Podemos implementar una alternativa sencilla a Celery usando directamente las tecnologías subyacentes a esta biblioteca. RabbitMQ es una cola de mensajes, idónea para la distribución de tareas entre nodos, mientras que Redis es una base de datos NoSQL de tipo clave-valor en memoria para el almacenamiento de los resultados y su posterior recuperación.

La ventaja de implementar directamente la funcionalidad de una cola de tareas es que el tiempo de inicio del worker en Celery es de aproximadamente un segundo, mientras que esta alternativa lo reduce a casi cero. Esto es una ventaja mínima en el caso de que el servicio permanezca arrancado y escuchando en el nodo, pero hay otra alternativa de diseño en que se ejecuta un worker en cada contenedor y se cierra al terminar la tarea, y en este caso ese ahorro de tiempo es muy importante.



4.3. Localización del cliente

Una vez elegido qué tipo de cliente se va a ejecutar en el sistema distribuido, hay que definir en qué lugar va a estar situado.

4.3.1. Workers dentro de los contenedores

En el *host* se levantan un número elevado de contenedores, todos ellos gestionados con Kubernetes o con *restart policy* [10], y dentro del propio contenedor se ejecuta el cliente que obtiene de algún modo el código que se tiene que ejecutar.

POLÍTICA DE REINICIO (RESTART POLICY)

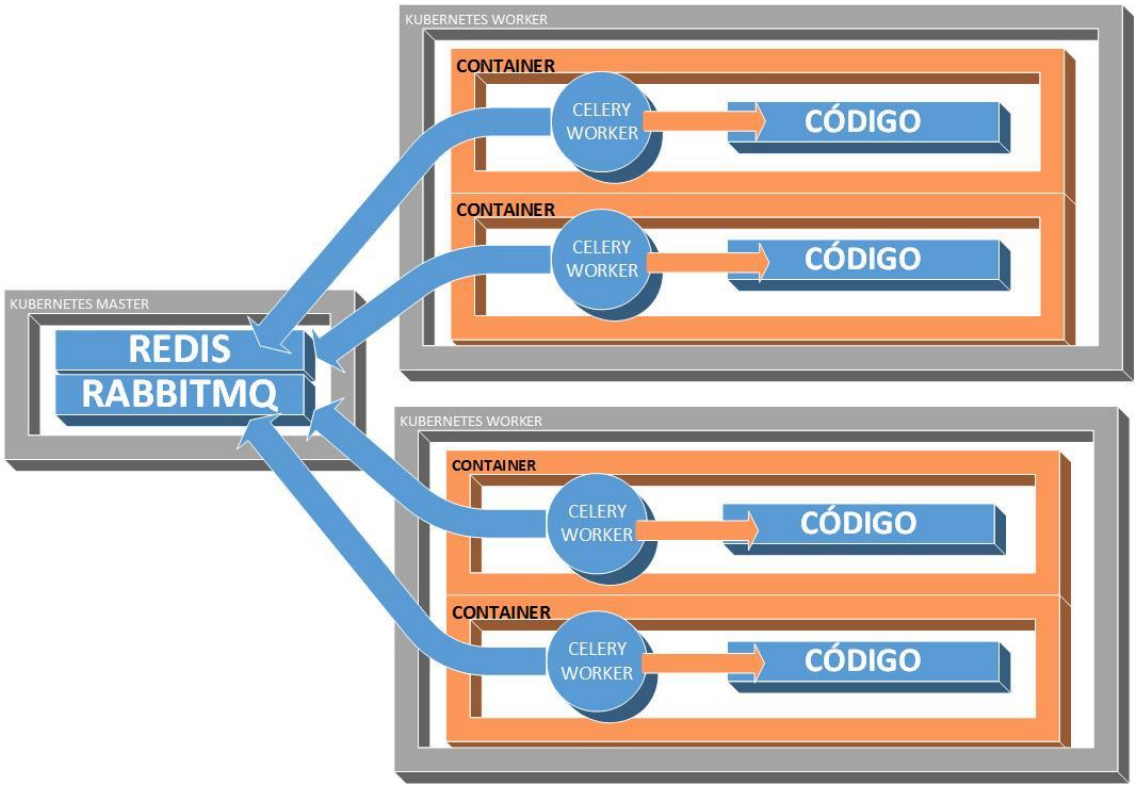
Cada contenedor Docker tiene asociada una política de reinicio, que define qué sucede cuando el contenedor es cerrado o falla. Existen cuatro políticas determinadas por Docker:

- no: el contenedor no vuelve a arrancar bajo ninguna circunstancia
- on-failure: sólo se reinicia en caso de fallo
- unless-stopped: se reinicia siempre que no haya sido cerrado explícitamente
- always: se reinicia siempre

Si la política elegida es unless-stopped o always, el contenedor será arrancado incluso si el host fue reiniciado.

Este método tiene algunos problemas, principalmente hay cuellos de botella. El tiempo de arranque de cada contendor con un test de 4 hilos, realiza 20 lanzamientos por hilo y tarda desde 0,5 segundo a 5,5, con una media de 1,8 segundos y su cierre dependiendo si el contenedor tiene algo ejecutándose puede llegar a tardar unos 10 segundos. Cada arranque de Celery es un segundo aproximadamente. Por último, el cierre de Celery es de un segundo dependiendo del comando que se use para el cierre. A este problema se le añade que si un contenedor se cierra antes de los 10 segundos Kubernetes lo asume como un fallo e intenta reiniciar el contendor.

Si la opción es lanzar el Dispatcher [20] manual con RabbitMQ y Redis, se gana tiempo en los cuellos de botella, pero la política de balanceo es *first come, first serve* [7]. Pero los problemas como el cierre prematuro siguen existiendo en los contenedores.



4.3.2. Workers fuera de los contenedores

Un proceso en el Dispatcher arranca y mantiene un número de contenedores para su uso exclusivo (no gestionados por ningún otro mecanismo) y cuando recibe una tarea utiliza uno de estos contenedores ya arrancados para ejecutarla.

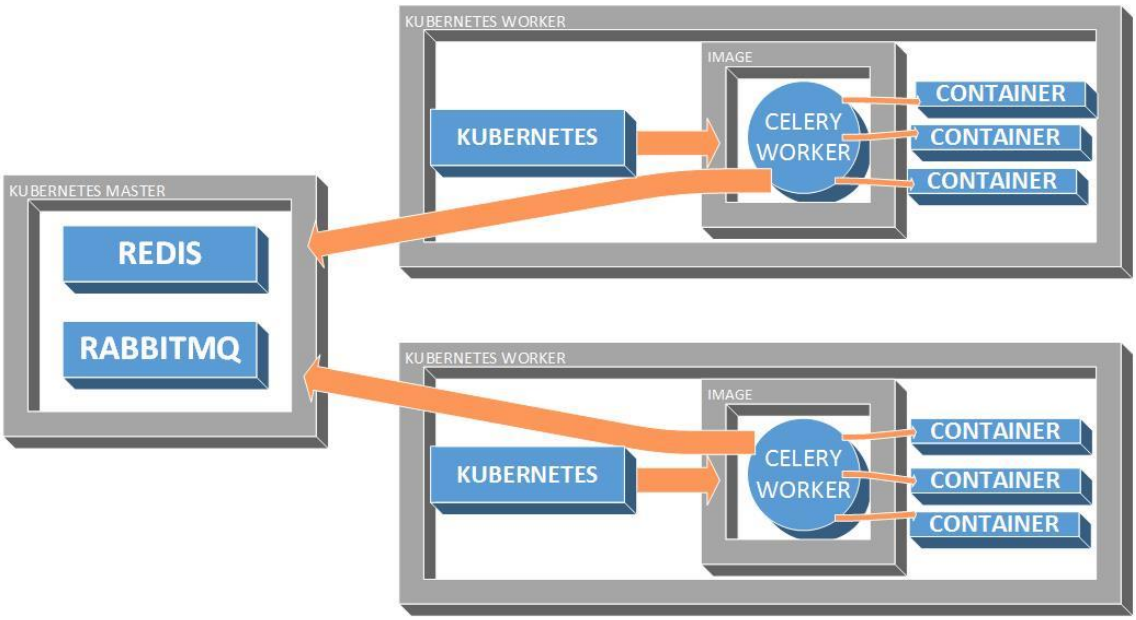
Celery lanza un contenedor cada vez que recibe una petición, en cuestión de tiempo es un problema que antes hemos comentado. Cada lanzamiento de Celery tarda un segundo aproximadamente, al igual que el lanzamiento de cada contenedor y si las tareas que se quieren ejecutar en cada contenedor son rápidas. Se volverá a asumir un cierre prematuro e intento de reiniciar el contenedor, esto implica que se tarde mucho tiempo en la ejecución.

POOL [18] Y PREFORK [13]

Cuando la creación de una nueva instancia de algún tipo de objeto o servicio consume demasiado tiempo y recursos, es habitual intentar crear dichos recursos cuando el sistema no está ocupado, de tal manera que cuando el recurso se necesita ya está creado y disponible. Un conjunto de recursos disponibles para el sistema que están esperando a ser utilizados se denomina *pool*.

Cuando los recursos instanciados son procesos, a la acción de crear varios procesos y dejarlos en espera se suele llamar pre-lanzamiento o *prefork*.

Si el container hace pool dentro de un contenedor Docker gestionado por Kubernetes, requiere una ejecución un tanto especial para que el demonio [17] Docker del host pueda ser invocado desde dentro del contenedor, el dispatcher tiene que introducir el contenido de la petición en un archivo `tar` [19] y convertirlo en `BASE64` para hacer llegar la petición al nodo ejecutor.



5. Implementación

Llegado al punto de la implementación hay que decidir qué tecnologías van a ser utilizadas.

Parte común

La comunicación entre el dispatcher y los ejecutores es la primera decisión que debe tomarse, y en este caso la alternativa entre servicios web y una cola de mensajes es sencilla de dirimir en nuestro caso. La cola de mensajes tiene una ventaja determinante: permite que los clientes se bloqueen en espera en lugar de tener que realizar espera activa de forma mucho más sencilla para el desarrollador, lo que conlleva menos espacio para cometer errores.

Elegida la opción de la cola de mensajes, se plantea la utilización de una opción que facilita su utilización aún más si cabe: Celery. Esta biblioteca es relevante ya que nos ofrece reintentos automáticos si el mensaje no se confirma, tiene la opción de confirmar antes y después de la ejecución, además proporciona una cola de tareas, RabbitMQ, para que los worker recojan las peticiones que el master ha dejado y una base de datos NoSQL, Redis, para que los workers almacenen sus respuestas una vez acabada la compilación del código. Además, Celery nos ofrece un balanceo de carga basado en una arquitectura interna de "prefetch", que consiste en cada worker se queda con más tareas de la que está ejecutando, siendo útil para evitar un tráfico constante, no tienen que pedir una nueva tarea cada vez que termina una. Celery se trata de una librería de Python por lo que estamos limitando el sistema a la utilización de un solo lenguaje. También en este caso las ventajas son claras, si bien hay que tener muy en cuenta el mayor inconveniente: el código deberá cambiarse para excluir Celery si se decide no utilizar Python o no llamarlo desde otro lenguaje.

PREFETCH

Cuando un cliente obtiene tareas de una cola puede hacerlo de una en una o tomando varias a la vez, lo que se denomina *prefetching*. La principal ventaja de esta técnica es que, sobre todo con mensajes de gran tamaño o que necesitan relativamente mucho tiempo para ser leídos y/o decodificados, se reduce la latencia de lectura al solapar la ejecución de una tarea con la lectura de otras de la cola.

El *prefetch* sirve además como un método rudimentario de balanceo, evitando que un sólo nodo acapare todas las tareas.

La siguiente decisión es la localización del cliente. Como ya hemos visto, podemos lanzar un número de contenedores dentro de los cuales se ejecuta un cliente en cada uno, o bien podemos lanzar un solo cliente que se encargue de lanzar y cerrar

contenedores, así como de insertar el código en cada uno de ellos, ejecutarlo y tomar el resultado cada vez que llegue una petición. La decisión en este caso es más complicada, y para tomarla nos hemos basado en benchmarks del tiempo necesario para lanzar un *worker* de Celery así como para crear un nuevo contenedor y cerrarlo. Un worker de Celery sin opciones especiales tarda un segundo en arrancar, número que permanece estable, aunque se lancen 20 workers a la vez, aunque se puede reducir hasta unos 0.2 segundos con las opciones "`--without-gossip --without-mingle`". Sin embargo, el lanzamiento de un contenedor varía dependiendo del número de lanzamientos concurrentes que se intenten hacer. Un único lanzamiento tarda de 0.5 a 0.8 segundos, pero si se utilizan tres hilos independientes que lanzan contendores a la vez, el tiempo máximo de lanzamiento sube hasta los 5 segundos. Se produce un fenómeno similar en el cierre de los contenedores: el cierre de un sólo contenedor es casi instantáneo pero el cierre en paralelo de varios implica un retraso de un segundo.

Estas cifras nos llevan a diseñar un mecanismo que nos permita tener múltiples contenedores abiertos y utilizarlos cuando lleguen peticiones; lo que se suele denominar "*prefork*". Este sistema lanza y cierra los contenedores en un hilo independiente y no obstaculiza ni bloquea el hilo principal, por lo que los tiempos de lanzamiento y cierre quedan enmascarados, siempre que en el caso del lanzamiento haya contenedores disponibles en el *pool* cuando llega una petición.

La construcción de este pool de contenedores conlleva la existencia de un único cliente por host, no un cliente por contenedor.

La última decisión es el uso de herramientas de orquestación para gestionar las imágenes que contienen los clientes. En este punto indicamos dos variantes, debido a que el código ha sido desarrollado para soportar ambas. La decisión sobre cuál variante escoger depende de las características de las máquinas en las que se va a ejecutar este sistema, así como de las necesidades de uso.

Variante A: con orquestación

En primer lugar, decidir cuál va a ser la tecnología que se encargue de orquestar el clúster, debido a la cantidad de documentación y estar en el ranking de tecnologías más usadas tanto en el sector empresarial, como en el sector particular. Kubernetes es la elegida para orquestar el sistema, dicho clúster tiene un nodo master y 5 nodos worker, en los que se debe instalar Docker y ejecutar el código que nos proporciona el git de Kubernetes para poder lanzarlo, dependiendo si el nodo es worker o master. En este paso se descubrió un problema, el nodo master tenía ocupado el puerto 8080, y kubernetes, por defecto, se levanta en el puerto 8080, generando conflictos y no dejando que se inicie correctamente. Como solución se descargó el código de Kubernetes, se modificó y se volvió a subir para que se lanzará en el puerto 28080.Una vez instalado Kubernetes utiliza Flannel [24] para la conexión entre nodos. A la hora de lanzar un contenedor Kubernetes y Docker tienen una política de cierre prematuro,

que está relacionada con la “Docker restart policy”, que contiene cuatro políticas de reinicio, “no” (por defecto), “on-failure”, “unless-stopped” y “always”. Si el cierre del contenedor se produce antes de 10 segundos desde el arranque, se considera como un fallo y se aplica un “exponential backoff” [12], que vuelve a intentar lanzar el contenedor.

EXPONENTIAL BACKOFF

En general, *exponential backoff* es un algoritmo que disminuye la frecuencia de cierta operación hasta encontrar un punto aceptable.

Kubernetes implementa exponential backoff en el reinicio de un contenedor. Si un contenedor recién lanzado se cierra antes de 10 segundos, K8s interpreta que ha habido algún tipo de error y retrasa su reinicialización en lugar de efectuarla inmediatamente, esperando más tiempo cada vez para el reinicio de dicha máquina hasta llegar al número máximo de reintentos.

Una vez montado el sistema distribuido hay que ajustar el sistema a los requisitos, con la configuración ulimit de docker a la hora de lanzar las imágenes, añadiendo a la hora de lanzar las imágenes con algunas features, “`--cpus`” para controlar el número de CPUs que queremos usar para lanzar la imagen, “`--stop-timeout`” tiempo en segundos que tendrá de vida el contenedor. Uno de los requisitos consiste en que cada contenedor sólo puede ejecutar un código para evitar programas malintencionados, la parada y rearranque implican mantener datos en el sistema de archivos, por lo que la única solución es el cierre de cada contenedor después de ejecutar un código y lanzar uno nuevo para cada petición.

Variante B: sin orquestación

Otra opción es tener el mismo sistema, pero sin un orquestador, ya que Celery funciona como Kubernetes: en el nodo master se ejecuta Celery y distribuye sus tareas entre los nodos workers en los que se ha ejecutado el código de Celery Worker.

6. Manual de instalación

En este capítulo se va a detallar una guía para la instalación de un sistema distribuido para la ejecución segura de código. La instalación de Kubernetes es opcional, se puede desplegar Celery igualmente.

LINUX CGROUPS

Cgroups es una característica del kernel de Linux cuyo fin es controlar y limitar los recursos asignados a grupos de tareas. En casi todos los sistemas operativos tipo Unix existe un mecanismo, *ulimit*, usado para limitar recursos por cada proceso. En contraste, Cgroups permite aplicar estos límites a un grupo de procesos, que en el caso de Docker está compuesto por los procesos que se ejecutan centro de cada contenedor. Además, hay un mayor número de límites que se pueden imponer a estos procesos.

Cgroups está dividido en módulos que pueden venir por defecto activos o no activos dependiendo de la distribución de Linux que estemos utilizando. En el caso concreto de estas instrucciones nos hemos basado en Debian, por lo que para poder utilizar algunas de las características de Cgroups y que Docker funcione correctamente, hay que pasar ciertos parámetros al kernel a través del cargador de arranque.

Kubernetes

- Previo a la instalación debemos tener un nodo master y tantos nodos workers como queramos.
- Instalación de Kubernetes, para este paso necesitaremos ser root.

1. Primero en todas las máquinas modificamos el grub.

```
Editar: nano /etc/default/grub

Poner: GRUB_CMDLINE_LINUX="cgroup_enable=memory"

Ejecutar: update-grub2

Reiniciar: reboot
```

2. En todas las máquinas (Sólo si se tuviera que repetir los pasos, la primera vez no es necesario).

```
docker stop $(docker ps -a -q) # Parar todas las instancias

docker rm $(docker ps -a -q) # Eliminar instancias

docker rmi $(docker images -q) # Eliminar imagenes
```

3. En la máquina Master.

```
git clone https://github.com/kubernetes/kube-deploy

cd kube-deploy/docker-multinode

export K8S_VERSION=v1.5.6

export IP_ADDRESS=192.168.134.1 # La IP de este equipo en la red

./master.sh
```

4. En todas las máquinas Worker.

```
git clone https://github.com/kubernetes/kube-deploy

cd kube-deploy/docker-multinode

export K8S_VERSION=v1.5.6

export MASTER_IP=192.168.134.1 # La IP del master

export IP_ADDRESS=192.168.134.1 # La IP de este equipo en la red

./master.sh
```

5. Kubectl, Sólo en el nodo Master.

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl

chmod +x ./kubectl

mv ./kubectl /usr/local/bin/kubectl
```

6. Comprobación, en el nodo Master que es el único que tiene instalado Kubectl.

```
kubectl cluster-info #En master nos debería dar toda la información del Clúster

kubectl get nodes #En master para consultar los nodos
```

7. Lanzar un replication controller desde un yaml.

```
Creamos un fichero, por ejemplo, en la ruta:
/opt/kubernetes/examples/nginx-pod.yaml
Contenido del archivo:


apiVersion: v1
kind: ReplicationController
metadata:
      # Nombre del Replication Controller
      name: my-nginx
spec:
      replicas: 1
      selector:
      app: nginx
      template:
      metadata:
            name: nginx
            labels:
                  app: nginx
      spec:
            containers:
                  - name: nginx
                  image: nginx
                   ports:
                     - containerPort: 80
                  restartPolicy: Always
```

Para lanzar el pod:

```
kubectl create –f /opt/kubernetes/examples/nginx-pod.yaml
```

Cliente

Es prerrequisito tener Docker instalado tanto en el servidor como en los clientes. En los clientes, además, deberá instalarse la biblioteca Celery con soporte para Redis:

```
pip3 install celery[redis]
```

El lanzamiento del cliente es muy sencillo. Sólo es necesario obtener el código en el servidor y en los clientes, por ejemplo, clonando el repositorio que lo contiene:

```
git clone https://github.com/velazq/executor-service.git
```

Deben lanzarse los servicios auxiliares (RabbitMQ y Redis) en el servidor:

```
cd executor-service

./start-services.sh
```

En los clientes, en cambio, ha de ejecutarse el worker:

```
cd executor-service

celery worker -A celery_worker
```

7. Pruebas de concepto y medidas de tiempos

Hemos creado un programa de prueba en Python que genera el valor N de la sucesión de Fibonacci usando el algoritmo recursivo, sin ningún tipo de optimización. Para esta prueba calcularemos la posición 48, que en un portátil tarda esto:

```
$ time python3 fib.py
63245986

real  0m23.552s
user  0m23.442s
sys   0m0.040s
```

Hemos generado 200 carpetas simulando el envío de prácticas al servidor. Todas ellas se encuentran bajo la carpeta "*codetest*".

Lanzamos el comando:

```
time python3 executor-service/celery_client_multi.py codetest
fib.py
```

y obtenemos como resultado:

```
real  3m22.149s
user  0m0.704s
sys   0m0.084s
```

8. Conclusiones y trabajo futuro

Este proyecto involucra muchas tecnologías diferentes tales como *containerización*, orquestación, comunicación entre procesos (IPC), protocolos de serialización, colas de mensajes o almacenes de valores clave.

La tecnología central de este proyecto, la *containerización* con Docker, se está acercando rápidamente a la madurez, aunque algunas peculiaridades todavía se mantienen. Su supuesta creación instantánea no es así en nuestras pruebas, probablemente debido a algún cuello de botella en el *daemon* Docker. Hay preocupaciones sobre el nivel de aislamiento que proporciona y si es suficiente para detener a un atacante determinado. Pero, por otro lado, su funcionalidad y rendimiento es inigualable. La virtualización real impone una sobrecarga pesada que Docker elimina casi por completo.

La orquestación, por el contrario, sigue siendo inmadura. Incluso la tecnología más prometedora, Kubernetes, que es desarrollada por Google y tiene el mayor número de usuarios, carece de documentación y facilidad de uso. El concepto subyacente es sólido y sin duda se convertirá en un lugar común en unos años, pero en este momento está implementación todavía está limitada a devotos equipos con suficientes recursos para mantener y depurar esta nueva tecnología.

IPC es una tecnología muy madura. Hay varias implementaciones con diferentes conjuntos de características y características de rendimiento, y la documentación es, en la mayoría de las alternativas, abundante. Nuestra elección, Celery, está bien documentada y tiene muchas opciones de configuración para ajustar su rendimiento, e incluso puede trabajar con muchas colas de mensajes y bases de datos diferentes, para la distribución y almacenamiento de tareas.

Este trabajo puede ser ampliado o utilizado como base para varias otras ideas. El más directo sería una plataforma de prueba para nuevos lenguajes de programación o DSL (lenguajes específicos de dominio), para que los usuarios pudieran ser presentados con una interfaz web donde pudieran escribir o cargar sus pruebas y hacerlas evaluar por el sistema.

Este proyecto también podría ser utilizado para implementar una de las últimas tendencias en computación: un proveedor sin servidor. Al igual que PaaS (plataforma como servicio) e IaaS (infraestructura como servicio), algunos proveedores como Amazon y Google están comenzando a ofrecer un servicio basado en eventos donde los desarrolladores suben su código y se ejecuta en respuesta a una llamada externa tal como un usuario que hace una solicitud a una página web o una base de datos que está siendo modificada. Estos servicios sólo se facturan por el uso de la CPU y por lo tanto no requieren tener una máquina virtual funcionando en todo momento, por lo que es más rentable para ciertos tipos de cargas. Otra ventaja es que estos servicios se

escalan automáticamente para el desarrollador, porque el código se puede ejecutar en cualquier número de máquinas en paralelo.

7. Conclusions and future work

This project involves many different technologies such containerization, orchestration, inter-process communication (IPC), serialization protocols, message queues or key-value stores.

The central technology of this project, containerization with Docker, is quickly approaching maturity, albeit some quirks still remain. Its purported instant creation is, in our tests, not so, probably due to some bottleneck in the Docker daemon. There are concerns about the level of isolation it provides and whether it is enough to stop a determined attacker. But, on the other hand, its functionality and performance is unparallelled. Real virtualization imposes a heavy overhead that Docker eliminates almost completely.

Orchestration, in contrast, is still immature. Even the most promising technology, Kubernetes, that is developed by Google and has the largest number of users, is severely lacking in documentation and ease of use. The underlying concept is sound and will no doubt become commonplace in a few years' time, but right now this implementation is still constrained to devops teams with enough resources to maintain and debug such a new technology.

IPC is a very mature technology. There are several implementations with different feature sets and performance characteristics, and documentation is, in most alternatives, plenty. Our choice, Celery, is well-documented and has many configuration options to adjust its performance and it can even work with many different message queues and databases, for task distribution and storage.

This work can be extended or used as the basis for several other ideas. The most straightforward one would be a testing platform for new programming languages or DSLs (domain-specific languages), so that users could be presented with a web interface where they could write or upload their tests and have them evaluated by the system.

This project could also be used to implement one of the newest trends in computing: a serverless provider. In the same fashion as PaaS (platform as a service) and IaaS (infrastructure as a service) some providers such as Amazon and Google are beginning to offer an event-driven service where developers upload their code and it gets executed in response to an external event such as a user making a request to a web page or a database being modified. These services are only billed for CPU usage and therefore do not require having a virtual machine running at all times, thus making them more cost-effective for certain types of loads. Another advantage is that these

services are scaled automatically for the developer, because the code can be run on any number of machines in parallel.

8. Contribución al proyecto

En este apartado se detalla la aportación de cada uno de los componentes al proyecto.

Jonathan Sánchez Paredes

Web Services

Cómo hemos detallado en el plan de trabajo, el primer punto a investigar fue qué tipo de web services se podría usar para la comunicación de los nodos y el transporte de las tareas.

Las dos primeras opciones surgieron rápidamente, una de ellas exponer el nodo Master, como servidor y el resto de nodos *Worker* como clientes, la otra opción, todos los nodos *Worker* como servidores y el nodo Master como cliente. En estas opciones se descubrió que la espera de los clientes era una espera activa y su consumo de recursos es muy elevado por lo que se descartó y se pasó a buscar otras opciones.

La siguiente opción que no realizaba una espera activa era un *Websocket*, que el nodo Master estaba comunicado con cada uno de los nodos *Worker* bidireccionalmente, es un método que tiene una cantidad grande de conexiones abiertas. *Websocket* era el método que teníamos pensado implementar hasta que descubrimos Celery.

Celery es una librería para Python que nos ofrece a parte de la conexión entre nodos, una cola de mensajes y una cola de tareas, en este caso son Redis Y RabbitMQ, sin tener conocimientos previos simplemente exportando las librerías.

Orquestación

Una vez elegido el tipo de conexión, que se va a utilizar, empecé a investigar qué es la orquestación, y qué tecnologías existen para ello. En el ámbito de los contenedores software, se entiende como orquestación a la gestión unificada y automatizada de dichos contenedores. El software de orquestación recibe una configuración deseada y se encarga de llevarla a cabo en los nodos que gestiona, levantando contenedores de imágenes específicas cuando es necesario y reiniciando si es preciso, incluso pueden aparte de organizar el sistema aplicar restricciones. como es el uso de CPU, tiempo de lanzamiento de un contenedor, límite de disco y muchas más restricciones que nos ayudan a cumplir los requisitos necesarios para nuestro sistema.

En cuanto a las tecnologías que existen para orquestar un sistema son innumerables pero la gran mayoría de ellas son tecnologías muy inmaduras, menos de un año o dos en el mercado, por lo que nos hizo descartar muchas de ellas. Tras una investigación las tecnologías más maduras y más usadas en el mercado son Kubernetes, Docker Swarm y Mesos con Marathon. La herramienta más madura y con mayor potencial es Mesos con Marathon (Mesos es el software y Marathon lo complementa con el orquestador) pero utiliza una gran cantidad de recursos, por lo

tanto, para la envergadura de nuestro sistema no era compatible, en cambio Kubernetes y Swarm ofrecen también un buen rendimiento, aunque Swarm viene instalado con Docker, Kubernetes está un grado por encima de Swarm, es mucho más usado y tiene una documentación mucho más completa.

Imágenes Docker

El siguiente paso son las imágenes Docker, cómo funcionan y para qué sirven, su finalidad es crear contenedores ligeros y portables para las aplicaciones software que puedan ejecutarse en cualquier máquina con Docker instalado, sin depender del sistema operativo. Para crear una imagen necesitamos un archivo llamado DockerFile, en que indicamos que necesitamos para la imagen:

```
FROM ubuntu:14.04

RUN apt-get update

RUN apt-get -qqy install git
```

Usamos docker build sobre el archivo para crear la imagen. Para el uso de las imágenes, docker nos ofrece un repositorio para ello, para bajarnos las imágenes hacemos pull y para subirlas push. Por último, para lanzar una imagen tenemos que hacer docker run -ti.

Instalación KBN y Modificaciones

Una vez hemos investigado el funcionamiento de todos los pasos para crear un sistema distribuido toca llevarlo a la práctica. Nuestro sistema comienza con un Clúster de 1 nodo Master y 5 nodos Worker, en el que decidimos instalar Kubernetes.

Es necesario tener instalado en todas las maquina Docker, y clonar el repositorio de Kubernetes en cada uno de los equipos. Para nuestra sorpresa Kubernetes utiliza el puerto 8080 y en el clúster dicho puesto estaba ocupado, como solución hubo que modificar el código de Kubernetes para que este se lanzará en el puerto 28080 y volverlo a subir, para poder instalarlo en todos los equipos.

Por otro lado, a la hora de Kubernetes bajarse las imágenes para sus conexiones, ocupaba todo el espacio de disco que tenía asignado y hubo que realizar un enlace simbólico a otro disco con más espacio.

Una vez instalado se comprobó si funcionaba correctamente lanzando pods, service o controller desde archivos yaml y habilitando WebUi para ver gráficamente donde eran lanzados los diferentes servicios.

Por último, comprobar si se podían lanzar imágenes en diferentes pods y como lo balanceaba y como se podía hacer externo un servicio.

Memoria

Para terminar el proyecto, quedaba redactar todo lo investigado y realizado hasta la finalización del proyecto, desde el principio se documentó cada investigación realizada para que no faltará ningún detalle. Se generó una estructura de memoria, en la que se pudiera trasladar todo lo que habíamos hecho hasta la fecha. Por último, realizar la presentación para exponer todas las decisiones tomadas y sus motivos.

Alberto Velázquez Alonso

Colas de mensajes y tareas

La investigación al respecto se centró en encontrar una alternativa a los microservicios, debido a sus características; rápidamente llegamos al concepto de la cola de mensajes y sus beneficios asociados. El siguiente paso fue investigar qué diferentes alternativas libres existen, y llegamos a RabbitMQ, Kafka, Beanstalkd y StormMQ.

Después de una comparación de todas ellas, la conclusión fue que el mejor equilibrio entre velocidad, funcionalidad, facilidad de instalación y de uso era RabbitMQ. Esta cola de mensajes dispone de paquetes precompilados para la mayoría de distribuciones de Linux, así como para MacOS. Sin embargo, y dado que el proyecto está basado fundamentalmente en Docker, tenía sentido ver si era posible ejecutar esta cola de mensajes dentro de un contenedor, y vimos que no sólo era posible, sino que existían ya imágenes preparadas a tal efecto.

Durante la subsecuente investigación sobre bibliotecas para usar RabbitMQ desde diferentes lenguajes de programación encontramos Celery, una biblioteca para Python que implementa funcionalidades extra y utiliza RabbitMQ internamente. Celery es en realidad una cola de tareas, que es una ampliación del concepto de cola de mensajes. Esto añade interesantes características que convienen a nuestro proyecto, ya que lo que de hecho estamos realizando es el procesamiento de una serie de tareas.

Debido a su uso por Celery, también investigamos la utilización y el funcionamiento interno de Redis, una base de datos no relacional que conserva los datos en memoria, lo cual la hace extraordinariamente rápida para consultas sencillas.

Uso de Docker

Docker es la tecnología en la que se basa todo el proyecto. El objetivo de esta tecnología es la creación de un entorno de ejecución aislado del resto de procesos en una máquina, pero sin la merma de rendimiento que suponen las máquinas virtuales tradicionales. La investigación al respecto se dividió en dos fases: su funcionamiento interno, y la utilización de esta tecnología.

Primero se investigó la instalación de Docker en una máquina con sistema operativo Debian GNU/Linux, especialmente las modificaciones requeridas en

componentes de dicho sistema operativo, como por ejemplo en el cargador de arranque y en el sistema de usuarios y permisos. Este paso se documentó con el fin de poder repetir dicha instalación.

En paralelo se estudió el funcionamiento interno de Docker: cuáles son sus conceptos fundamentales, en qué tecnologías se apoya, cómo se realizan las operaciones más frecuentes, cómo se utiliza desde un lenguaje de programación y cómo ejecutar de forma correcta las operaciones que necesitábamos hacer para este proyecto.

Este conocimiento se aplicó en un clúster del departamento compuesto por un master y cinco workers; en todos los nodos se encontraba instalado Docker. En este clúster se realizaron los tests de rendimiento y las pruebas de código.

Pruebas preliminares con código

Conociendo las funcionalidades básicas de las tecnologías implicadas, el siguiente paso fue desarrollar diferentes proyectos de prueba para medir el rendimiento de las posibles soluciones, así como la facilidad de uso y la robustez de la solución. En este punto se tomaron buena parte de las decisiones arquitectónicas, descartando las tecnologías que no se adaptaban a nuestras necesidades y estudiando más en profundidad la utilización de aquellas que sí lo hacían.

Tests de rendimiento de Docker

En este punto del desarrollo del proyecto descubrimos algunas características del rendimiento de Docker, en concreto la ralentización en los procesos de arranque y cierre de contenedores, cuando se lanzaban varias peticiones simultáneas a Docker. Esto nos llevó a realizar algunas pruebas algo más detalladas que evidenciaron que, efectivamente, dichos problemas existían. Esta evidencia finalmente llevó al desarrollo de una solución para mitigar esta ralentización arrancando varias máquinas al inicio de la aplicación y realizando estas operaciones de arranque y parada en hilos independientes.

Código: diversas opciones

Incluso habiendo descartado varias tecnologías encontramos que había razones para implementar soluciones diferentes, lo cual llevamos a cabo utilizando para ellos varios proyectos creados en GitHub, la plataforma abierta de control de versiones y desarrollo colaborativo de software.

Una de estas razones es el uso opcional de Kubernetes, que simplifica la gestión de contenedores, pero complica la instalación inicial y por tanto implica una decisión que los administradores del sistema están mejor preparados para tomar que nosotros.

Otra razón es la interoperabilidad, ya que Celery es una biblioteca que proporciona una serie de funcionalidades que simplifican muy notablemente el código, pero sólo funciona en Python, por lo que existe una alternativa utilizando directamente RabbitMQ y Redis.

Memoria

El último paso ha sido recopilar todas las investigaciones y pruebas en este documento y en una presentación, estructurando las diferentes líneas de investigación en apartados ordenados y sencillos de leer, con descripciones de las diferentes tecnologías y explicaciones razonadas de cuáles han sido las diferentes alternativas planteadas, así como los motivos para elegir una u otra.

9. Bibliografía

Kube-deploy

https://github.com/kubernetes/kube-deploy

Instalar Kubernetes

https://severalnines.com/blog/installing-kubernetes-cluster-minions-centos7-manage-pods-services

Instalar Kubernetes en diferentes sistemas operativos

http://www.tothenew.com/blog/how-to-install-kubernetes-on-centos/

https://kubernetes.io/docs/getting-started-guides/fedora/fedora_manual_config/

http://tdeheurles.github.io/how-to-run-local-kubernetes/

https://kubernetes.io/docs/getting-started-guides/scratch/

Following Cgroup subsystem not mounted: [memory]

https://github.com/kubernetes/kubernetes/issues/26038

http://container-solutions.com/trying-out-kubernetes/

Docker compose

https://github.com/ContainerSolutions/kubernetes-demo/blob/master/docker-compose.yml

Creación y ejecución de yaml y explicación de controller

https://www.adictosaltrabajo.com/tutoriales/primeros-pasos-con-kubernetes/

kubernetes la tecnología con mayor proyección http://thenewstack.io/tns-research-present-state-container-orchestration/

Comparison of containers schedulers

https://medium.com/@ArmandGrillet/comparison-of-container-schedulers-c427f4f7421

Container Orchestration Wars

https://www.youtube.com/watch?v=C_u4_l84ED8

Kubernetes vs Docker Swarn

https://platform9.com/blog/compare-kubernetes-vs-docker-swarm/

Guide Kubernetes

https://deis.com/blog/2016/kubernetes-illustrated-guide/

Kubernetes architecture and use

https://medium.com/@itmarketplace.net/kubernetes-101-12ad2424d2f1#.t65x0chdm

http://blog.kubernetes.io/2016/09/kubernetes-1.4-making-it-easy-to-run-on-kuberentes-anywhere.html

https://deis.com/blog/2016/kubernetes-the-hard-way/

Kubernetes pods

http://kubernetes.io/docs/user-guide/production-pods/

Kubernetes container

http://kubernetes.io/docs/user-guide/pods/single-container/

FLOP

http://dl.acm.org/citation.cfm?id=2401807

10. Glosario

- [1] Micro-servicio: La arquitectura de microservicios es un patrón por el cual una aplicación se divide en módulos o fragmentos que se comunican entre sí mediante peticiones HTTP.

- [2] Endpoint: Es el destino final de una conexión o aplicación.

- [3] REST: Es un estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web.

- [4] Cola de mensajes: Una cola de mensajes tiene características adicionales a una cola normal. Su función es la distribución de mensajes entre procesos, y permite que los clientes bloqueen cuando la cola está vacía. Tiene otras características interesantes como permitir que los clientes se suscriban sólo a ciertos mensajes y no a todos (ver publisher / subscriber).

- [5] Cola de tareas: Una cola de tareas es, al menos conceptualmente, una extensión de la cola de mensajes que además requiere que, al recibirse una tarea, los clientes lo notifiquen.

- [6] Push / Pull: En un escenario en el que hay productores y consumidores, el evento en el que un consumidor solicita datos a un productor se denomina *pull*, mientras que, si es el productor el que inicia la comunicación de los datos disponibles a los consumidores o les señaliza dicho evento de alguna forma, se denomina *push*. Cada una de las dos variantes tiene diferentes características de rendimiento.

- [7] First come, first serve: El primero en llegar es el primero en salir, como en una fila.

- [8] In memory database: Almacena todos sus datos en memoria. Esto implica que las consultas son mucho más rápidas al no tener que leer ni escribir en un disco, pero como contrapartida el riesgo de pérdida de datos es mucho más elevado al no existir persistencia de datos.

- [9] Key-value store: Al contrario que una base de datos relacional, una *key-value store* almacena pares clave-valor y no permite realizar búsquedas complejas, o al menos no tan complejas como SQL

- [10] Política (de reinicio): “Docker restart policy”, que contiene cuatro políticas de reinicio, “no” (por defecto), “on-failure”, “unless-stopped” y “always”.

- [11] Clúster: Es un grupo de máquina que están relacionadas y se comunican en busca de un objetivo.

- [12] Exponential backoff: Si el cierre del contenedor se produce antes de 10 segundos desde el arranque, se considera como un fallo y se aplica un “exponential back-off”, que vuelve a intentar lanzar el contenedor.

- [13] Prefork: Cuando los recursos instanciados son procesos, a la acción de crear varios procesos y dejarlos en espera se suele llamar pre-lanzamiento o *prefork*.

- [14] Balanceo: Es un concepto usado en informática que se refiere a la técnica usada para compartir el trabajo a realizar entre varios procesos, ordenadores, discos u otros recursos.

- [15] Orquestación: En el ámbito de los contenedores software, se entiende como *orquestación* a la gestión unificada y automatizada de dichos contenedores.

- [16] Container: Los contenedores no son más que "cajas" aisladas con lo esencial para poder ejecutar un determinado programa o aplicación. Eso se puede entender como una máquina virtual ligera, en vez de las completas y pesadas con las que se trabaja en la virtualización completa.

- [17] Demonio: servicio o programa residente es un tipo especial de proceso informático no interactivo, es decir, que se ejecuta en segundo plano en vez de ser controlado directamente por el usuario.

- [18] Pool: Un conjunto de recursos disponibles para el sistema que están esperando a ser utilizados se denomina *pool*.

- [19] TAR: Se refiere en Informática a un formato de archivos ampliamente usado en entornos UNIX, identificados por el sufijo de archivo .tar

- [20] Dispatcher, executor, task: El nodo Dispatcher es el nodo que recibe la tarea "task", y la manda a un nodo executor para que la ejecute.

- [21] Espera activa, poll: Quedarse en espera constantemente hasta recibir la petición.

- [22] Nodo, host: Un host o anfitrión es un ordenador que funciona como el punto de inicio y final de las transferencias de datos. Comúnmente descrito como el lugar donde reside un sitio web. Un anfitrión de Internet tiene una dirección de Internet única (dirección IP) y un nombre de dominio único o nombre de anfitrión (hostname).

- [23] Websocket: Es una tecnología de comunicación bidireccional desarrollada primeramente para navegadores web, pero que en realidad es independiente de HTTP.

- [24] Flannel: Es una red virtual que proporciona una subred a cada host. Kubernetes asume que cada pod tiene una IP única y enrutable dentro del clúster. Reduce la complejidad de realizar asignaciones de puerto.

- [25] K8s: Es una abreviación de Kubernetes derivada de las 8 letras de "ubernetes" en 8.

- [26] RKT: Fue creado tras encontrar diversos fallos de seguridad en Docker, este nuevo sistema ofrece seguridad en las imágenes del contenedor, previene ataques de escalado de privilegios, más flexibilidad en la publicación de imágenes y portabilidad a otros sistemas de "containerización".

- [27] FLOP: Es un software que sigue la metodología Test Driven Design, el uso de este programa está destinado a la programación y el aprendizaje. Este software alberga problemas de programación, permite al profesor añadir fácilmente otros nuevos y también evalúa automáticamente las soluciones enviadas por los alumnos.