

FACULTAD DE INFORMÁTICA

ADRIÁN GARCÍA GARCÍA


Arquitecturas Cloud y microservicios

30 de abril de 2017



This work is licensed under a [CC-BY-SA 4.0 License](https://creativecommons.org/licenses/by-sa/4.0/).

Índice

1. Introducción	2
2. Arquitectura orientada a servicios	4
2.1. Comunicación	5
2.2. Microservicios	6
3. Docker 	8
4. DevOps	9

Resumen

Este trabajo tiene como objetivo exponer y profundizar en los conceptos expuestos en una conferencia optativa que tuvo lugar durante la semana de la informática de 2017. Los ponentes fueron dos ingenieros de la empresa GMV (Ricardo de Castro y Roberto Galán). En concreto, el nombre de la conferencia era el siguiente: Despliegue automático de arquitecturas escalables basadas en microservicios sobre el Cloud de Google (23 de Febrero, 11-14 horas). Conviene puntualizar que al final no usaron la plataforma de Google, sino que se basaron en Amazon Web Services y Docker Swarm para desplegar una aplicación web que se basaba en el uso de microservicios para su funcionamiento. Al mismo tiempo, hablaron de un concepto que está muy de moda últimamente como es el enfoque *DevOps*, también está relacionado con el despliegue rápido y eficiente de código en la nube, la automatización de procesos de desarrollo (*testing*, recogida de métricas, despliegue, etc.) y las metodologías ágiles.

1. Introducción

Durante la conferencia, se hizo especial hincapié en las necesidades de los clientes y las características que estos exigen en una aplicación de carácter empresarial que se ejecuta en la nube. Los requisitos más destacados son los siguientes:

- **Fiabilidad.** El funcionamiento de las aplicaciones web está íntimamente relacionado con la imagen que las grandes empresas tienen en la sociedad, si una tienda de comercio electrónico se cae durante una hora puede suponer pérdidas millonarias para la empresa y acarrea el riesgo de que sus clientes dejen de confiar en ella y se pasen a la competencia. Debido a esto, los clientes esperan que su aplicación tenga el mínimo número de errores posibles. Uno de los aspectos más importantes en este sentido es que se haga un proceso meticuloso de *testing* y validación antes de publicar una aplicación, priorizando las partes más críticas del sistema y que pueden suponer un cuello de botella para la misma.
- **Recuperación instantánea ante fallos** (*Zero Down Time*[3]). En caso de que la aplicación llegue a fallar, lo que es imposible de evitar al completo, es necesario minimizar el tiempo en el que no se provee

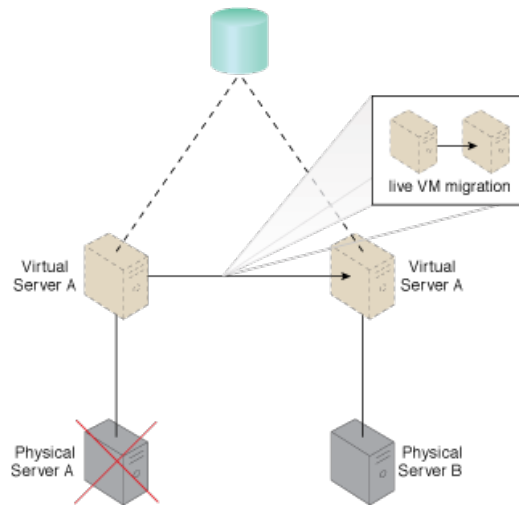


Figura 2: Esquema de recuperación ante fallos.

un servicio. Asegurar un tiempo mínimo de recuperación supone un desafío importante cuando una máquina física actúa como el punto de fallo único de la aplicación. Sin embargo, el hecho de tener una aplicación desplegada en contenedores virtuales, nos facilita en gran medida asegurar un sistema tolerante a fallos. Este sistema debe constar de ciertas características, entre las que se destacan:

- Disponer de un **cluster de máquinas** configurado.
- Aplicar técnicas de **replicación y balanceo de carga**.
- Tener montado un **volumen compartido de datos** que permita a diferentes máquinas acceder a las imágenes que utilizan las máquinas virtuales. Por ejemplo, *Network File System (NFS)*.

Cómo se puede observar en la Figura 2, cuando el servidor físico A falla, se desencadena una migración del contenedor virtual al servidor B. Lo que debería ocurrir sin complicaciones en caso de que el volumen de datos compartido por red siga disponible y solo haya ocurrido un problema aislado en la máquina A. En general, el demonio encargado de vigilar el correcto funcionamiento de los nodos se denomina *watchdog* y su funcionamiento se basa en el intercambio periódico de mensajes con los nodos del cluster para intercambiar información del estado del servicio (*keepalive o heartbeat*).

- **Tiempo rápido de despliegue.** Uno de los problemas más extendidos en el ámbito de las aplicaciones empresariales de elevada complejidad es que la aplicación puede tener muchas dependencias entre diferentes componentes, lo que hace difícil realizar un despliegue limpio y sin errores. Se ponía como ejemplo el clásico problema de que en el entorno de desarrollo funciona todo perfectamente, pero cuando se traslada a producción deja de hacerlo y cada equipo le echa la culpa a otro.
- **Escalabilidad.** Los servicios pueden recibir grandes picos de tráfico y deben ser capaces de responder ante los mismos sin errores. Es evidente que la optimización de cada componente de la aplicación es un aspecto importante, pero además se debe diseñar un sistema capaz de balancear la carga y aumentar la dedicación de recursos replicando sus unidades de cómputo para responder ante los incrementos en demanda. Esto es relativamente fácil de hacer en entornos cloud o con soluciones de virtualización como Docker.

2. Arquitectura orientada a servicios

El concepto de creación de arquitecturas orientadas a servicios, comúnmente conocido como SOA (**Service Oriented Architecture**), trata de definir aplicaciones basadas en el uso de una **colección servicios que pueden trabajar de forma autónoma**, pero que también intercambian mensajes y se coordinan para cumplir una funcionalidad más compleja. Se parte del concepto de servicio, que es una funcionalidad bien definida, autocontenida y que no depende del contexto o estado de otros servicios. Por ejemplo, validar un cliente en la aplicación o proveer datos sobre el funcionamiento de la misma. La división en servicios tiene las ventajas de que la realización de *tests* y la corrección de errores se puede realizar de una forma mucho más precisa, y no es necesario cambiar la aplicación completa para corregir fallos o introducir nuevas funcionalidades.

Para implementar una arquitectura orientada a servicios de forma eficiente, es necesario tener en cuenta una serie de conceptos y estrategias de diseño:

- Tener claro un **conjunto de servicios** que se quieran proveer a nuestros clientes o incluso de forma interna en la organización.

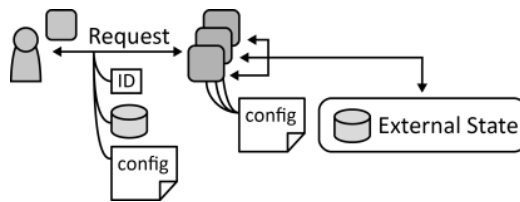


Figura 3: Peticiones del cliente con estado almacenado fuera de los componentes[6].

- Añadir una serie de procesos de **orquestación**, **balanceo de cargas** y de **recolección de métricas** para asegurar el buen funcionamiento de los mismos.
- Aplicar un conjunto de patrones de diseño que permita realizar un **diseño modular**, que tenga una buena **encapsulación** de los componentes, que asegure el **aislamiento**, la **reusabilidad** y la **escalabilidad**.
- Utilizar un modelo de programación compatibles con diferentes estándares de comunicación y que sea soportado por múltiples tecnologías, por ejemplo, los **servicios REST**.

2.1. Comunicación

En una arquitectura orientada a servicios uno de los aspectos más importantes es la comunicación, ya sea entre los propios componentes de nuestra aplicación o los métodos de nuestra API que exponemos al público. Los servicios de una aplicación distribuida pueden estar alojados en diferentes máquinas, lo que hace necesario que el estado de nuestra aplicación sea almacenado de forma externa a los componentes, estos se conocen como **Stateless Components** (puede verse un esquema en la Figura 3). Una de las técnicas más habituales para implementarlos es **incluir la información de estado en las peticiones** que recibe cada componente, otra forma de hacerlo sería transmitirla en los mensajes *keepalive* enviados por los procesos de orquestación.

En este contexto que requiere una comunicación entre múltiples componentes sin ninguna comunicación previa de esquemas de datos, como si era

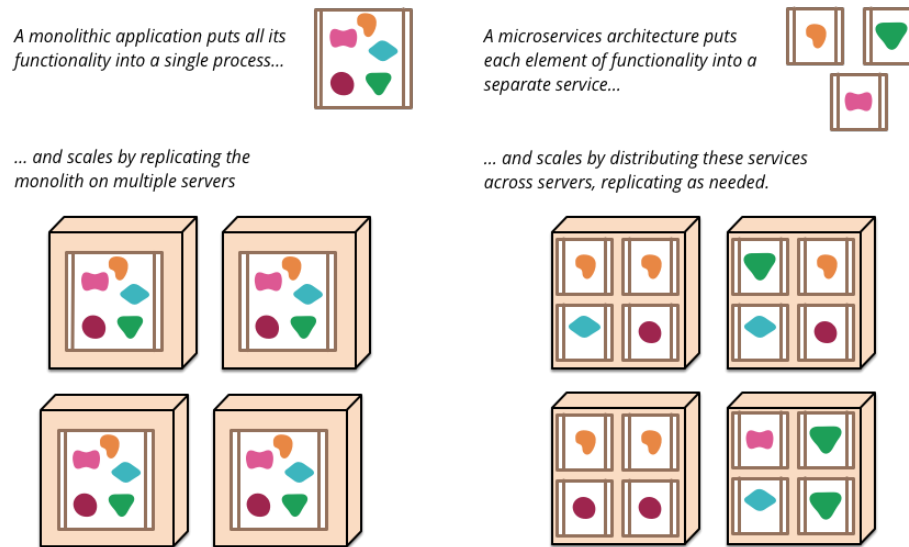


Figura 4: Aplicación monolítica VS microservicios.

necesario con SOAP o XML, podemos destacar la utilidad de las APIs REST por las siguientes características:

- Representan un **protocolo cliente/servidor sin estado**.
- **Facilitan la portabilidad** y la comunicación entre diferentes plataformas y lenguajes, ya que se basan en el **protocolo HTTP** y el uso de sus verbos: POST, GET, PUT y DELETE.
- Permiten **exponer recursos de forma sencilla** diferentes mediante las URI (Unique Resource Identifier).
- Ofrecen una gran **flexibilidad para transmitir diferentes tipos de datos**: html, json, text, xml, jpeg, etc.

2.2. Microservicios

Las aplicaciones monolíticas se despliegan como un solo bloque de front-end, lógica de negocio y acceso a datos. Si existe un cuello de botella en sólo un elemento del sistema necesitaríamos desplegar varias instancias de

la aplicación completa y aplicar algún tipo de balanceo de carga para poder mejorar la latencia de respuesta. Esta es una solución pésima, ya que desperdicia recursos y no resulta eficiente para incrementar el rendimiento.

Como se puede apreciar en la Figura 4, las aplicaciones diseñadas usando microservicios permiten escalar aquellas partes de la aplicación que representen un cuello de botella. Se basa en el concepto de arquitecturas SOA pero dividida aún más en cada dominio (Productos, Empleados, Ventas...). Los componentes pueden ser atómicos o compuestos, que agrupan varios dominios.

Las principales ventajas que aportan el uso de microservicios son las siguientes:

- La división de las aplicaciones en microservicios nos permite hacer una mayor **reutilización de componentes** en otras aplicaciones. Varios proyectos suelen compartir una lógica de negocio o de acceso a datos similar, y el diseño modular de los microservicios nos aporta una gran flexibilidad para adaparlos a otros problemas.
- Como ya mencionamos anteriormente, aporta una gran **mejora en la escalabilidad del sistema** en función de nuestras necesidades.
- El proceso de **mantenimiento se simplifica** enormemente, ya que los componentes sin estado interno permiten reducir el aconchamiento y es más sencillo detectar y resolver los problemas en un único componente sin detener al completo el servicio.
- El sistema es **resistente a fallos** ya que, a diferencia de las aplicaciones monolíticas, el fallo de un componente no supone necesariamente el fallo de la aplicación.

Desplegar suele resultar el paso más complicado en este tipo de aplicaciones que en las arquitecturas monolíticas convencionales, es necesario levantar múltiples servicios que suelen presentar interdependencias y, por lo tanto, requieren que se realice un orden de arranque concreto. Existe una serie de servicios de código abierto que permiten gestionar los diferentes microservicios y las interacciones que presentan. Algunos de los más usados pertenecen al framework Spring Cloud de Netflix:

- Eureka: Service Discovery.

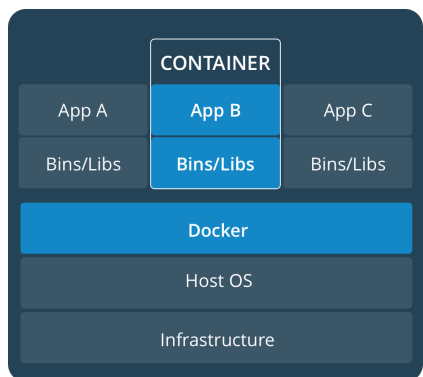


Figura 5: Infraestructura de un contenedor.

- Zuul: Proxy.
- Ribbon: Load Balancer.

3. Docker

Es una tecnología de virtualización que permite definir aplicaciones auto-contenidas. Se basa en el concepto de contenedores, se tratan de entornos de ejecución portables y aislados de la máquina anfitriona. Estos contenedores incluyen todas las librerías y binarios necesarios para el despliegue de una aplicación. Esto permite definir contenedores ligeros de forma eficiente, que tienen todo lo necesario para que una aplicación se ejecute y que tras su despliegue no se oiga la típica excusa de: 'En mi máquina funcionaba'.

A diferencia de las máquinas virtuales convencionales, las tecnologías de contenedores como Docker no incluyen un sistema operativo completo en su imagen, tan solo las librerías y configuraciones necesarias para su funcionamiento (se puede observar un esquema en la Figura 5). Esto permite mejorar la eficiencia al no existir la sobrecarga que supone la interacción de las máquinas virtuales con el sistema operativo huésped(guest), ya que Docker es capaz de realizar una interacción directa con el Kernel Linux de la máquina anfitrión(host).

Algunas características destacadas sobre Docker en la conferencia son las siguientes: es portable, ligero, automatizable, autocontenido, aislado(ofrece entorno de desarrollo que se comporta a modo de *sandbox*), agiliza el despliegue y el mantenimiento.

Con muchos contenedores pueden aparecer problemas de coordinación, para solucionarlos se utilizan balanceadores de carga o administradores (también conocidos como orquestadores). Algunos de los más conocidos son:

- Kubernetes (Google)
- Amazon Web Services ECS
- Docker Swarm

4. DevOps

Otro de los conceptos introducidos en la conferencia es el de DevOps (el término proviene del acrónimo inglés formado por *Development y Operations*), es un concepto que ha atraído un interés creciente en el ámbito de la gestión de proyectos y la ingeniería del Software durante la última década. Recibe una gran influencia de las metodologías ágiles de desarrollo y propone una mejora de la comunicación entre los equipos de desarrollo y operaciones. A lo largo de la historia, se ha producido un conflicto entre ambos equipos que surge de forma natural de sus roles, en el área de desarrollo se intenta introducir nueva funcionalidad mientras que el equipo de operaciones crea estabilidad y se ocupa de resolver errores.

Las técnicas de DevOps se basan en la creación de una cultura común entre ambos equipos, que se aproveche al máximo de la automatización de procesos, la obtención de métricas que permita guiar la toma de decisiones y la integración de ambos equipos de forma que participen en todo el ciclo de vida del proyecto(diseño, desarrollo, mantenimiento. . .).

Una de las herramientas más utilizadas para automatizar las actividades del ciclo de vida del Software es Jenkins. Esta herramienta permite definir tareas para automatizar el proceso de compilación, *testing*, despliegue y recopilación de métricas. Otras herramientas que pueden resultar de utilidad son SonarQube(Testing de calidad de código) y NexusOSS(Gestor de dependencias y artefactos).

Referencias

- [1] **Cloud Computing Patterns, Springer (2014)**. *Fundamentals to Design, Build, and Manage Cloud Applications*. Authors: Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeak and Peter Arbitter.
- [2] [Fundaments of Cloud Service Reliability](#), Microsoft Secure Blog.
- [3] [Zero Downtime](#), cloudpatterns.org.
- [4] [Web Services Explained](#) (Communication). service-architecture.com.
- [5] [Service Oriented Architecture](#): simply good design. ibm.com.
- [6] [Stateless Component](#), cloudcomputingpatterns.org
- [7] [API REST: ventajas](#). bbvaopen4u.com.