

FACULTAD DE INFORMÁTICA

ADRIÁN GARCÍA GARCÍA


Arquitecturas Cloud y microservicios

14 de mayo 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. Cloud Computing	4
3. Arquitectura orientada a servicios	5
3.1. Comunicación	6
3.2. Microservicios	7
4. Docker 	10
5. DevOps	11
6. Conclusiones	12

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 hacía 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, aunque se mencionaron brevemente, más tarde encontré diferentes enlaces con más información sobre el tema en [5]. 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* [6]). En caso de que la aplicación llegue a fallar, lo que es imposible de evitar

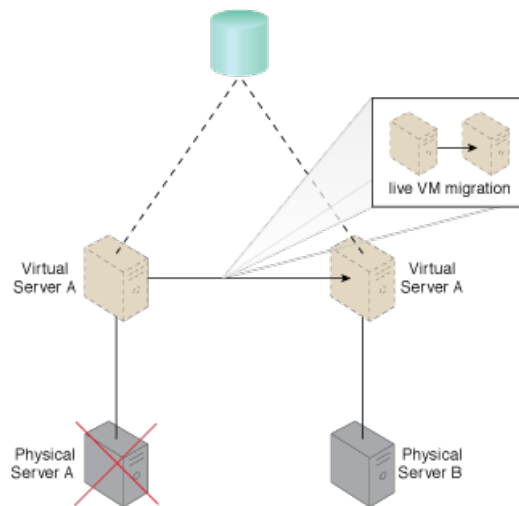


Figura 2: Esquema de recuperación ante fallos.

al completo, es necesario minimizar el tiempo en el que no se provee 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.

Este trabajo está estructurado de la siguiente forma, en la Sección 2 se realizará una exposición de diferentes conceptos que se utilizan en *Cloud Computing*. A continuación se introducirá el paradigma de las arquitecturas orientadas a servicios (SOA) en la Sección 3. Más adelante se introducirá la herramienta de contenedores Docker (Sección 4), con la que se pueden construir de forma sencilla diferentes tipos de aplicaciones basadas en microservicios. Tras esto, en la Sección 5 Para finalizar, en la Sección 6 presentaré una serie de opiniones personales que se han originado a partir de la conferencia y de la investigación posterior realizada.

2. Cloud Computing

A continuación se presentarán diferentes conceptos utilizados en el diseño y construcción de aplicaciones que se ejecutan servicios alojados la nube, concepto que se refiere a el alquiler de servidores o servicios específicos a un proveedor externo en función de nuestras necesidades de cómputo, este paradigma es comúnmente conocido como *Cloud Computing*. Algunos de las

propiedades esenciales que pueden variar de un proveedor de servicio a otro son las siguientes [2]:

- **Servicios bajo demanda.**
- **Acceso a conexión de alto ancho de banda.**
- **Disponibilidad de recursos.**
- **Elasticidad del servicio.**
- **Recolección de métricas.**

En función del modelo de servicio que nos ofrezca cada proveedor, podemos clasificar los servicios Cloud en 3 clases fundamentales:

Para más información sobre este tema y una visión completa de los patrones de diseño que se utilizan en Cloud Computing, se puede consultar este recurso [1].

3. Arquitectura orientada a servicios

El paradigma 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 por causa de una petición específica (Ej. Obtener datos del producto con ID 152). 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. Se puede obtener más información sobre los servicios web y su funcionamiento en la siguiente referencia [7].

Para implementar una arquitectura orientada a servicios de forma eficiente, es necesario tener en cuenta una serie de conceptos y estrategias de diseño [8]:

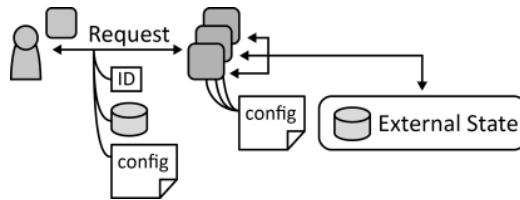


Figura 3: Peticiones del cliente con estado almacenado fuera de los componentes. Más información sobre componentes sin estado en [9].

- Tener claro un **conjunto de servicios** con unos objetivos bien definidos que se quieran proveer a nuestros clientes o incluso de forma interna a la organización.
- Añadir una serie de procesos de **orquestración**, **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 compatible con diferentes lenguajes de programación y que sea soportado por múltiples tecnologías y estándares de comunicación, por ejemplo, los **servicios REST**.

3.1. 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 sí era necesario con protocolos 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.

Se puede encontrar una descripción detallada de las APIs REST y sus ventajas en el blog de desarrolladores de BBVA [10].

3.2. Microservicios

Las aplicaciones convencionales que se han desarrollado tradicionalmente para problemas tecnológico de cierta envergadura se despliegan como un solo bloque monolítico de *frontend*, lógica de negocio y acceso a datos. Sin embargo, este diseño de arquitectura conlleva ciertas desventajas en el caso de necesitar escalar la capacidad de nuestra aplicación. En concreto, si existe un cuello de botella en sólo un elemento del sistema, necesitaríamos desplegar varias instancias completas de la aplicación 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.

El término de microservicios ha ganado una gran popularidad en los años recientes, se originó alrededor de 2014 y atrajo mucha atención hacia esta nueva forma de concebir la estructura de las aplicaciones. Puede aportar muchos beneficios si se hace uso del mismo para solucionar los problemas correctos y se comprende cuáles son sus ventajas y en qué casos sería mejor no usarlos. Uno de los riesgos que presenta construir una aplicación desde cero usando este patrón de diseño es que no sabemos exactamente cuales

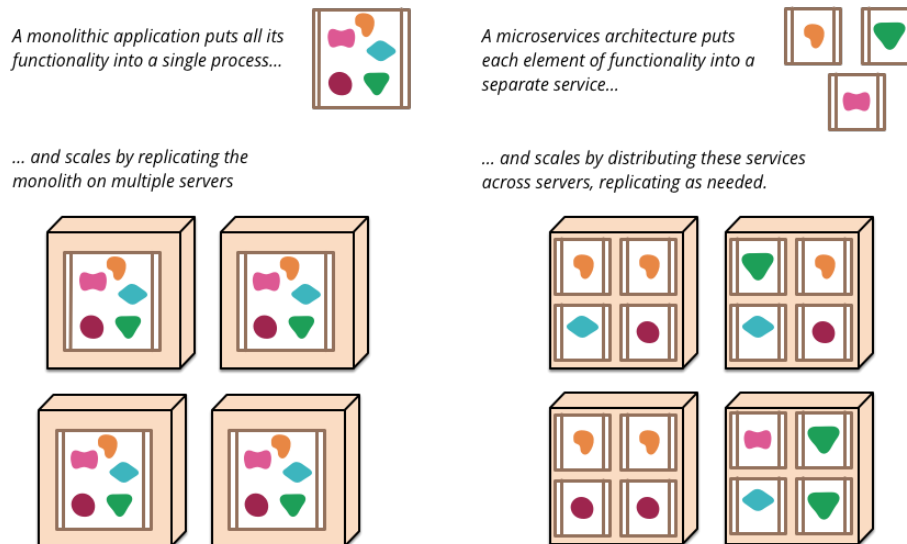


Figura 4: Aplicación monolítica VS microservicios.

son los requisitos de nuestra aplicación hasta que no empezamos a producir prototipos y recibir retroalimentación de los clientes, además los equipos con poca experiencia tienen la tendencia de querer usar nuevas técnicas de desarrollo como esta sin conocerlas en profundidad y sin saber si se pueden aplicar a su dominio. A pesar de todo, una gran cantidad de proyectos se ha beneficiado del diseño con microservicios y muchas compañías han obtenido buenos resultados al migrar aplicaciones o partes de las mismas a esta nueva técnica de diseño. Considero que uno de los principales motivos del éxito de los microservicios es que presenta una gran sinergia con el paradigma del Cloud Computing, y permite hacer un diseño modular de la aplicación que facilita de forma natural la división de trabajo en un reparto con objetivos claros y sencillos, sobre los que se puede hacer *testing* de forma sencilla y corregir errores sin afectar a otros componentes del sistema.

Las aplicaciones que tienen una arquitectura basada en microservicios, como se puede apreciar en la Figura 4, 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. Más información sobre la ejecución distribuida de aplicaciones en microservicios se puede encontrar en [11].

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 acoplamiento 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.

El proceso de despliegue suele resultar uno de los aspectos más complejos en este tipo de aplicaciones basadas en microservicios. El principal motivo es que, a diferencia de las arquitecturas monolíticas convencionales, es necesario levantar múltiples servicios que presentan dependencias específicas entre sí. Por lo tanto, requieren que se realice el arranque del sistema un orden concreto. Para facilitar estas tareas, existe una serie de servicios de código abierto que permiten gestionar los diferentes microservicios y las interacciones que presentan. Algunos de los más usados (presentados por los conferenciantes) pertenecen al framework Spring Cloud de Netflix:

- Eureka: Service Discovery.
- Zuul: Proxy.
- Ribbon: Load Balancer.

Cabe destacar el rol que pueden aportar servicios de virtualización como docker en este ámbito, ya que a partir de la versión 1.12 incluye un sistema de orquestación de contenedores, conocido como *Docker Swarm*, que permite definir y configurar la interconexión entre conjuntos de servicios de forma

sencilla con archivos de definición. Esta herramienta, que se explicará en la siguiente sección, es esencial para la construcción de aplicaciones basadas en microservicios ya que permite tanto el desarrollo en tu máquina local como la conexión con múltiples proveedores cloud y el despliegue de tu aplicación usando drivers específicos para los principales proveedores (Amazon Web Services, Google Compute Engine, Microsoft Azure, OpenStack, etc.). Esta forma de desarrollo en base a contenedores Docker, facilita la automatización de gran parte del ciclo de vida Software y está íntimamente relacionada con las técnicas de integración continua y DevOps (se explicarán detalladamente en la sección 5): desarrollo local, *testing*, integración de código de forma automática con servicios de control de versiones y despliegue en un servidor Cloud o privado mediante contenedores que tienen todas las dependencias necesarias satisfechas para que se ejecute la aplicación. Si queréis conocer a fondo los principios de diseño usando microservicios recomiendo un libro que me ha ayudado bastante a realizar este trabajo ya que trata la mayor parte de temas de los que he hablado, se trata de [4].

4. 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

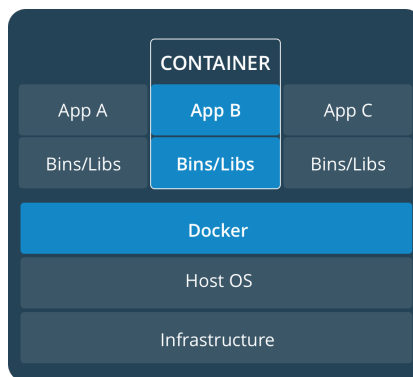


Figura 5: Infraestructura de un contenedor.

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

```
$ docker node inspect worker1 | grep IDdocker
node update --label-add type=master
18yrb7m650umx738rtevojpqy
```

5. 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(Análisis de calidad del código) y NexusOSS(Gestor de dependencias y artefactos).

6. Conclusiones

Referencias

- [1] **Cloud Computing Patterns, Springer (2014).** *Fundamentals to Design, Build, and Manage Cloud Applications*. Authors: Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck and Peter Arbitter.
- [2] **The NIST definition of cloud computing (2011).** *Mell P. and Grance T.*
- [3] **A view of cloud computing. Communications of the ACM (2010)***vol. 53, no 4, p. 50-58. ARMBRUST*, Michael, et al.
- [4] **Building Microservices (2015, 1st edition).** O'Reilly Media, Inc. *Designing fine-grained systems*. Author: Sam Newman.
- [5] [Fundaments of Cloud Service Reliability](#), Microsoft Secure Blog.
- [6] [Zero Downtime](#), cloudpatterns.org.
- [7] [Web Services Explained](#) (Communication). service-architecture.com.

- [8] [Service Oriented Architecture: simply good design.](#) ibm.com.
- [9] [Stateless Component,](#) cloudcomputingpatterns.org
- [10] [API REST: ventajas.](#) bbvaopen4u.com.
- [11] [Artículo sobre Microservicios y automatización.](#) Martin Fowler.