

FACULTAD DE INFORMÁTICA

ADRIÁN GARCÍA GARCÍA


Arquitecturas Cloud y microservicios

19 de junio 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	7
3.1. Comunicación	8
4. Microservicios	9
4.1. Gestión de datos descentralizada	12
5. Docker 	14
5.1. Manual práctico de Docker	15
6. DevOps	17
7. Conclusiones	17

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

En primer lugar, expondré diferentes conceptos que considero relevantes tanto en el campo del *Cloud Computing* como en el diseño de aplicaciones empresariales que se ejecutan en servicios web, que en general presentan un alto grado de complejidad y número de componentes.

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 [6]. 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.

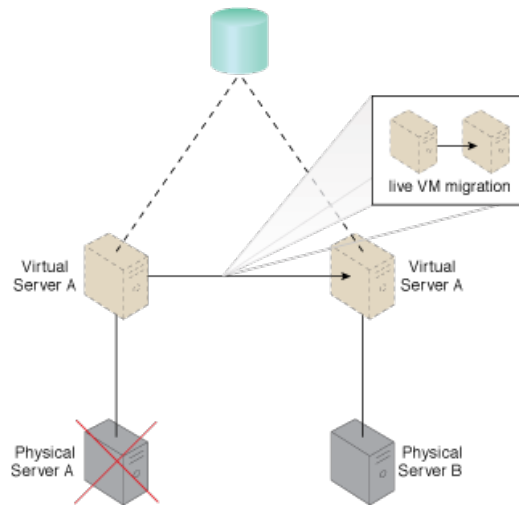


Figura 2: Esquema de recuperación ante fallos.

- **Recuperación instantánea ante fallos** (*Zero Down Time* [7]). 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 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 (*Single Point of Failure*). 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 5), con la que se pueden construir de forma sencilla diferentes tipos de aplicaciones basadas en microservicios. Tras esto, en la Sección 6 Para finalizar, en la Sección 7 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*. El Instituto

Nacional de Estándares y Tecnología de Estados Unidos (NIST) publicó en 2002 una serie de principios de diseño y definiciones de *Cloud Computing* [2]. Algunos de las propiedades esenciales que pueden variar de un proveedor de servicio a otro son las siguientes:

- **Servicio bajo demanda auto-escalable.** Un cliente obtiene unas capacidades de cómputo como tiempo de un servidor o almacenamiento en red en función de sus necesidades sin requerir interacción humana con el proveedor de servicio.
- **Disponibilidad de recursos.** Los recursos ofrecidos por el proveedor son ofrecidos a múltiples clientes, que comparten infraestructura pero cada uno dispone de una diferente asignación de recursos físicos y virtuales. Hay cierta sensación de incertidumbre en cuanto a la localización del servicio. Sin embargo, muchos proveedores permiten especificar a alto nivel la región, país o incluso *datacenter* en el que se ejecutará la aplicación. En general, los clientes pueden escoger entre diferentes planes de pago, que permiten adaptar a nuestras necesidades la capacidad de almacenamiento, el tipo (HDD vs SSD), la memoria RAM, el procesador, el ancho de banda o incluso la arquitectura en la que se ejecutará nuestra aplicación (cada vez más proveedores permiten que nuestra aplicación haga uso de GPUs y otros aceleradores).
- **Servicio de conexión multiplataforma con un alto ancho de banda.** Pongamos el ejemplo de una pequeña o mediana empresa (incluso una grande para un proyecto con necesidades variables a lo largo del tiempo), el coste de adquirir y mantener un conjunto de servidores propios para desplegar una aplicación en muchas ocasiones no deriva un beneficio suficiente para rentabilizar la inversión. Sin embargo, las capacidades del servicio ofrecidas a través de la red por los proveedores Cloud se adaptan a tus necesidades, tienen una calidad de servicio similar a la de un ISP (*Internet Service Providers*) y disponen de una infraestructura con un diseño de alta eficiencia energética. Además, se ocupan de toda la infraestructura y seguridad para que las aplicaciones sean accedidas mediante mecanismos estándar (HTTP) y en plataformas heterogéneas (móviles, tablets, portátiles, servidores, etc.).
- **Recolección de métricas.** Los sistemas Cloud controlan y optimizan automáticamente el uso de recursos a través de la recolección de diferentes métricas de rendimiento (ancho de banda, porcentaje de USO

de CPU y disco, cuentas de usuario activas, etc.). Además, se suelen aportar diferentes herramientas de visualización como los *dashboards*, que permiten a los usuarios tomar decisiones sobre el tipo de servicio que se ajusta más a sus necesidades.

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

- **Software as a Service (SaaS)**. Este modelo representa a los servicios prestados por diferentes aplicaciones que se ejecutan en una infraestructura Cloud. Las aplicaciones pueden ser accedidas simultáneamente por múltiples clientes a través de un programa o un navegador (Ej. GitHub o Google Drive). Sin embargo, el consumidor no gestiona ni tiene el control de la infraestructura Cloud subyacente.
- **Platform as a Service (PaaS)**. Los servicios prestados al cliente se basan en el uso de lenguajes, librerías, servicios y herramientas para crear y desplegar una aplicación en una infraestructura Cloud gestionada por el proveedor. En ocasiones el cliente puede ajustar la capacidad de cómputo y controlar ciertos parámetros pero no tiene permisos de *root* sobre la infraestructura. Algunos ejemplos son Microsoft Azure¹ o Google App Engine. La opción más destacada de código abierto en este ámbito se trata de Red Hat OpenShift.
- **Infrastructure as a Service (IaaS)**. Este tipo de servicio se centra en ofrecer a los clientes capacidad de procesamiento, almacenamiento, ancho de banda y otros recursos computacionales que pueden gestionarse a nivel del Sistema Operativo con control de *root* y que habitualmente se encuentran en un entorno virtualizado aislado del resto de clientes a los que se provee de servicio. Algunos ejemplos son Amazon EC2 o Google Compute Engine.

Por otra parte, se pueden diferenciar varios modelos de despliegue en función de las condiciones de propiedad bajo las que se ofrece el servicio. Si la infraestructura se provee para un uso exclusivo de una única organización se

¹Microsoft, al igual que Amazon o Google, ofrece un conjunto muy amplio de servicios que pueden ser clasificados en varias categorías. Sin embargo, se destaca la integración que Azure ofrece con las aplicaciones escritas en .NET lo que lleva a que el centro de sus servicios Cloud sean las aplicaciones en C#.

trata de *Private Cloud*, que puede ser propiedad de la organización o incluso ser gestionada por una diferente. También existe la posibilidad de que una comunidad de clientes que comparten ciertos objetivos (misión, requisitos de seguridad, políticas de cumplimiento de estándares, etc.) hagan uso de una misma infraestructura (*Community Cloud*). Además, existen múltiples casos de infraestructuras públicas que se comparten por organizaciones académicas, gobiernos o conjuntos de empresas (*Public Cloud*). Como se puede observar hay una gran variedad de modelos de propiedad y tipos de servicios; y, aunque existen múltiples alternativas, existen combinaciones de las diferentes clases que se conocen como *Hybrid Cloud*.

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 [8].

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

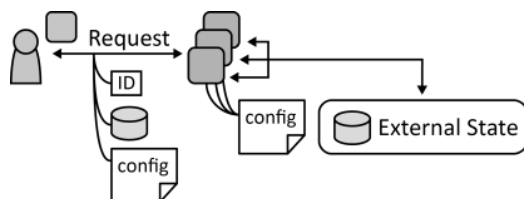


Figura 3: Peticiones del cliente con estado almacenado fuera de los componentes. Más información sobre componentes sin estado en [10].

- 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 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

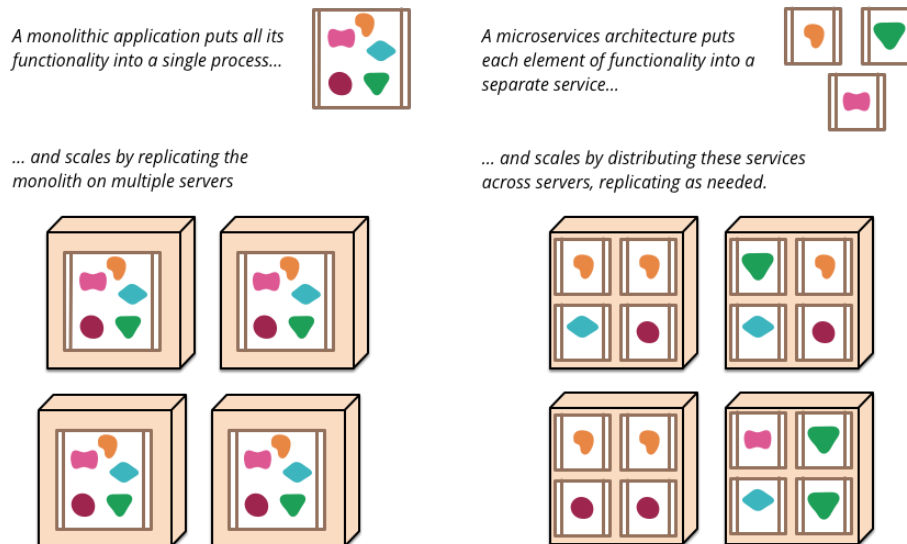


Figura 4: Aplicación monolítica VS microservicios.

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 [11].

4. 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.

Las arquitecturas basadas en microservicios se pueden definir como un estilo de desarrollo de aplicaciones que están formadas por conjuntos de servicios pequeños y con objetivos bien definidos, que se ejecutan en máquinas diferentes y emplean mecanismos de comunicación estándar y de baja latencia. Los microservicios habitualmente se exponen como un recurso a través de una API con verbos HTTP: GET, POST, etc. En general, cada microservicio responde a peticiones sobre un elemento en particular de la lógica de negocio y existen procesos de orquestación encargados de mantener el correcto funcionamiento del sistema y escalar el número de servicios disponibles en función de la demanda.

El concepto de diseño basado en microservicios ha ganado una gran atención en los últimos años, 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 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.

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, etc.). 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 [12].

A continuación expondré algunas de las ventajas e inconvenientes del uso de microservicios que considero más relevantes:

- + 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 adaptarlos 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.
- Los programas distribuidos son más complejos de programar y depurar. Las llamadas a procedimientos remotos pueden ser lentas y producir fallos. En este contexto resulta de gran utilidad el uso de tecnologías de paso de mensajes resistentes a fallos como RabbitMQ o Apache Kafka.
- Mantener la consistencia de datos en un sistema distribuido no es una tarea trivial, especialmente si se requiere el uso de diferentes tipos de bases de datos.
- El desarrollo de aplicaciones basadas en microservicios presenta cierta complejidad de organización, ya que requiere de un equipo capaz de coordinarse y gestionar un gran número de servicios que se re-despliegan regularmente.

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.). 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].

4.1. Gestión de datos descentralizada

La gestión de datos de forma descentralizada se presenta en diversas formas. A un nivel abstracto, se basa en la noción de que el modelo conceptual del mundo será diferente entre unos sistemas y otros. Este problema ocurre de forma habitual cuando se integran diferentes componentes en una gran compañía, el punto de vista de la gestión de productos y el carro de compra será muy diferente del sistema de recomendación o la gestión del soporte a los clientes. Mientras que las aplicaciones monolíticas prefieren una única base de datos relacional que incluso se comparten entre diferentes aplicaciones debido al elevado precio de las licencias, en el mundo de los microservicios es preferible que cada componente maneje su propia conexión con una base de datos,

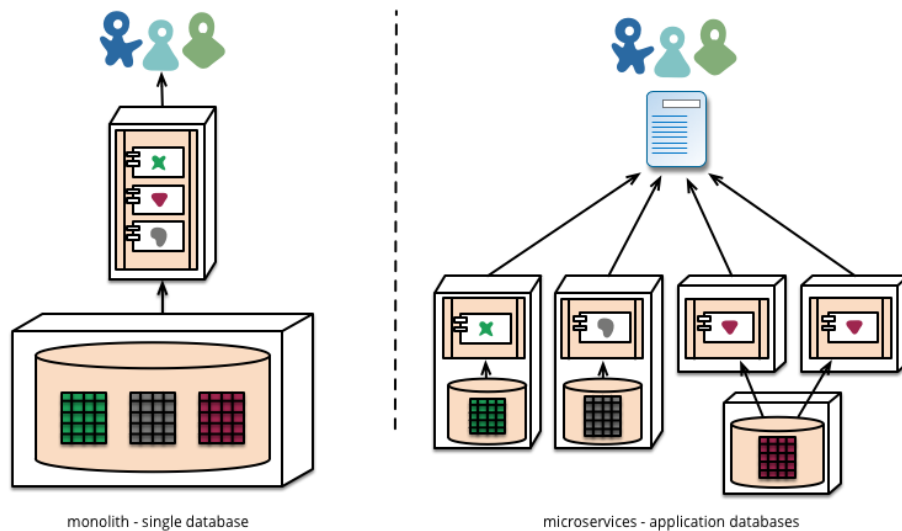


Figura 5: BBDD de aplicación monolítica VS persistencia políglota.

ya sean instancias de la misma base de datos o tecnologías completamente diferentes. Esta aproximación se conoce como **persistencia políglota** [13] y se puede observar una esquema de su arquitectura en la Figura 5.

Descentralizar la responsabilidad de la consistencia de datos entre diferentes servicios representa un reto para los desarrolladores, ya que se deben implementar diferentes técnicas que aseguren cierta atomicidad en las operaciones. Una posibilidad es la utilización de transacciones distribuidas, aunque impone ciertos problemas de acoplamiento al poder afectar a múltiples servicios y bloquear el sistema. En ocasiones, las necesidades de negocio llevan a preferir aceptar un cierto grado de inconsistencia para ser capaces de responder rápido a las peticiones.

La elección de tecnologías de almacenamiento no es un problema trivial de resolver, pero cada vez más la tendencia se dirige a usar ciertas bases de datos que brillan en resolver problemas concretos. Por ejemplo, las bases de datos noSQL aportan una gran flexibilidad para manejar catálogos de productos o colecciones de datos heterogéneos y no estructurados. En cambio, si queremos hacer un sistema de recomendación quizá sea más interesante disponer de un sistema de almacenamiento de datos que incorpore de forma nativa algoritmos de procesamiento de grafos (Neo4J). Una de las ventajas de los sistemas noSQL como MongoDB o Cassandra es que permiten su

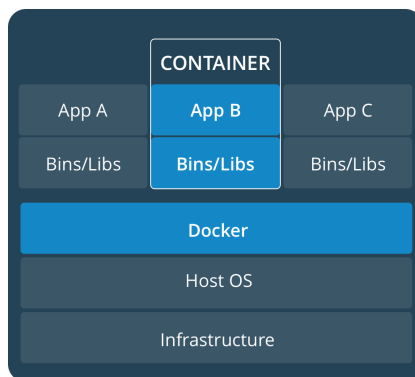


Figura 6: Infraestructura de un contenedor.

ejecución en clusters de máquinas distribuidas y son capaces de manejar volúmenes enormes de datos, distribuyendo la carga mediante replicación y *sharding*.

5. 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 6). 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

5.1. Manual práctico de Docker

Para poder crear un grupo de máquinas y que se comuniquen entre sí en primer lugar debemos aprender a definir una imagen. Los contenedores Docker se ejecutan en base a una imagen que puede ser descargada del repositorio oficial de contenedores, conocido como Docker Hub. Además, podemos definirla y construirla nosotros mismos usando cualquier imagen existente como plantilla y añadiendo los componentes y librerías que necesitemos (se especifican en un archivo `Dockerfile`). Un ejemplo de este archivo sería el siguiente:

```
FROM docker/whalesay:latest
RUN apt-get -y update && apt-get install -y fortunes
CMD /usr/games/fortune -a | cowsay
```

Podemos distinguir tres opciones fundamentales en este tipo de archivos:

- **FROM** nos permite especificar una imagen de Docker en la que basar la nuestra.
- **RUN** especifican diferentes comandos que necesitemos ejecutar para configurar el entorno de ejecución de nuestro contenedor (instalar librerías y dependencias).
- **CMD** especifican los comandos que debe ejecutar el contenedor una vez arranca y se ha configurado.

A continuación, es necesario construir la imagen que hemos definido a través del siguiente comando: `"docker build -t website ."`. Tras esto, se puede ejecutar el contenedor² usando el siguiente comando: `"docker run website"`.

Al igual que hemos definido la composición de una imagen personalizada a través de un archivo, es posible definir la estructura de una aplicación y su interconexión a través de otro archivo (con extensión `.yml` o `.yaml`) y la herramienta Docker Compose. Si partimos de la base que ya hemos definido una imagen llamada "website" que contiene la lógica de presentación y control de nuestra aplicación web, podemos levantar esta imagen junto con su base de datos redis y definir los puertos en los que se conectan de la siguiente forma:

```
website:
  build: .
  ports:
    - "5000:5000"
  volumes:
    - .:/code
  links:
    - redis
redis:
  image: redis
```

De esta forma, hemos definido una aplicación formada por dos contenedores. Uno llamado "website", que será construido a partir de un archivo `Dockerfile` que se encuentra en el directorio actual. También se configura el puerto en el que se expondrá la aplicación, el directorio de donde se añadirá el código de la misma y los otros contenedores con los que está enlazado; en este caso solo se relaciona con su base de datos redis, cuya imagen descargará del Docker Hub. Una vez hemos definido este archivo, podremos ejecutarlo con el comando `"docker-compose up"`, que buscará el archivo `docker-compose.yml` y ejecutará las imágenes que se indican en el mismo resolviendo las dependencias.

²Cuando ejecutamos una imagen, Docker buscará primero si el repositorio local de nuestra máquina contiene alguna con el nombre dado y si no intentará descargarla de Docker Hub.

6. 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).

7. Conclusiones

Me siento obligado a hacer una mención especial a los conferenciantes, ya que hicieron una exposición detallada tanto de la base teórica, como de la componente práctica del diseño de aplicaciones basadas en microservicios usando Docker, AWS y otros componentes de arquitecturas SOA de la suite de servicios de código abierto Netflix OSS. Realizaron una exposición muy interesante de los conceptos detrás de las arquitecturas basadas en microservicios y de diferentes técnicas que emplean las grandes empresas para automatizar el proceso de desarrollo. Aunque fue una charla de 3 horas, no se hizo nada larga y la gente estuvo atenta y participativa en todo momento. Me gustaría recomendar que en los próximos cursos del Máster en Ingeniería Informática se repita esta conferencia pero en el horario de clase, ya que me

ha parecido de las más interesantes y enriquecedoras de este año a pesar de haber sido una conferencia optativa de la semana de la informática.

Según mi juicio, 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 las aplicaciones 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.

Creo que la tecnología de virtualización basada en contenedores Docker ha sido una de las mejores herramientas de código abierto que han surgido en los últimos tiempos, considero crítica su capacidad para interactuar directamente con el kernel Linux (mejorando en gran medida el rendimiento de servidores que alojan aplicaciones de múltiples clientes). Además, creo que el diseño de aplicaciones basadas en microservicios con Docker facilita la automatización de gran parte del ciclo de vida Software, y está íntimamente relacionado con las técnicas de integración continua y DevOps. Las ventajas que me parecen más relevantes son:

- Simplifica el proceso de configuración del entorno de desarrollo, permitiendo generar de forma sencilla a través de archivos de definición todas las dependencias necesarias.
- La flexibilidad derivada de satisfacer todas las dependencias de una aplicación y poder desplegarla sin preocupación alguna en cualquier servidor me parece tremendamente útil
- Facilita el desarrollo en un entorno local de pruebas totalmente aislado.
- Permite la realización de pruebas y tests de unidad de funcionalidades bien definidas e independientes. Esto facilita en gran medida el reparto de trabajo entre diferentes equipos de desarrollo que pueden progresar en paralelo.
- La integración de código de forma automática con servicios de control de versiones con servicios como Jenkins y despliegue en un servidor Cloud o privado mediante contenedores que tienen todas las dependencias necesarias satisfechas para que se ejecute la aplicación.

De los materiales que he recogido, considero especialmente enriquecedores los artículos que he leído sobre aplicaciones que usan bases de datos

heterogéneas [13]. Especialmente el Framework Synapse [5], que permite la integración de diversas bases de datos en una misma aplicación y ofrece al mismo tiempo diferentes garantías de consistencia, transaccionalidad y atomicidad en las operaciones. Esta herramienta permite incluir interfaces para acceder a bases de datos tan diferentes como PostgreSQL, MongoDB, Cassandra o Neo4J. Creo que uno de los aspectos más importantes en el mundo del *Big Data* es la correcta elección del sistema de almacenamiento de datos para su posterior procesamiento, ya que cada uno responde a necesidades de negocio muy concretas.

En resumen, creo que he disfrutado haciendo este trabajo, he aprendido muchos trucos de LaTeX y sin duda esta ha sido una de las mejores conferencias a las que he acudido este año, esto me ha motivado enormemente para buscar información y completar la base de conocimiento que se transmitió en la misma. A mi parecer, las tecnologías relacionadas con el Cloud Computing tienen una enorme proyección de futuro y es el deber de cualquier ingeniero que se precie conocer las bases del diseño de aplicaciones que proveen servicios a millones de personas en Internet, y que no van a parar de crecer en el futuro inmediato.

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] **NIST definition of cloud computing (2011).** *Mell P., 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] **Synapse: a microservices architecture for heterogeneous-database web applications (2015).** *Viennot, Nicolas, et al. Proceedings of the Tenth European Conference on Computer Systems. ACM.*[Proyecto en Github](#). [Paper](#).
- [6] [Fundaments of Cloud Service Reliability](#), Microsoft Secure Blog.

- [7] [Zero Downtime](#), cloudpatterns.org.
- [8] [Web Services Explained](#) (Communication). service-architecture.com.
- [9] [Service Oriented Architecture](#): simply good design. ibm.com.
- [10] [Stateless Component](#), cloudcomputingpatterns.org
- [11] [API REST: ventajas](#). bbvaopen4u.com.
- [12] [Artículo sobre Microservicios y automatización](#). Martin Fowler's Blog.
- [13] [Artículo sobre persistencia políglota](#). Martin Fowler's Blog.