TECNOLOGÍAS DE LOS SISTEMAS DE INFORMACIÓN DE LA RED

Tema 4

Miquel Gómez Corral Ing. Informática ETSINF, UPV

Tema 4

Despliegue de servicios

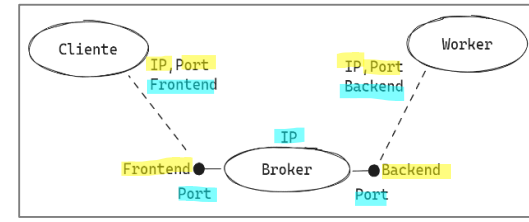
CONCEPTO DE DESPLIEGUE

Despliegue: Actividades que hacen que un sistema software esté preparado para su uso. **Instalación**, **activación**, **actualización** y **eliminación** de componentes o del sistema completo de manera automática sin tener que ir a mano.

Despliegue de una aplicación distribuida

Dado que una **aplicación distribuida** es una colección de componentes heterogéneos dispersos sobre una red de computadores, existen **dependencias** entre ellos (*porque cooperan*). Cada uno tendrá sus particularidades, requisitos y cosas de seguridad. *No puedes lanzar unos antes que otros etc.*

Patrón bróker-workers: la principal dependencia es hacer que un **cliente** sepa la ip/puerto del bróker-**frontend** y los **workers** sepan ip/puerto del bróker-**backend**. *Después, cada uno de ellos con un identificador para saber distinguirlo*.



Ya que puede haber varias instancias de cada componente, ha de poder iniciarse, actualizarse, desactivarse… de ***manera independiente del resto sin afectar al sistema****. Todo respetando las dependías y el orden entre ellos.*

DESPLIEGUE DE UN SERVICIO

Los sistemas distribuidos permiten ofrecer servicios (*funcionalidad*) a clientes: $Aplicación + Despliegue = Servicio$. ***Solo se le puede empezar a llamar servicio*** *cuando ya lo tienes todo hecho, resuelves dependencias y despliegas.* **Programa***: El código/fichero.* **Aplicación***: Eso funcionando.* **Servicio***: Son instancias, nodos ya furulando conectados.*

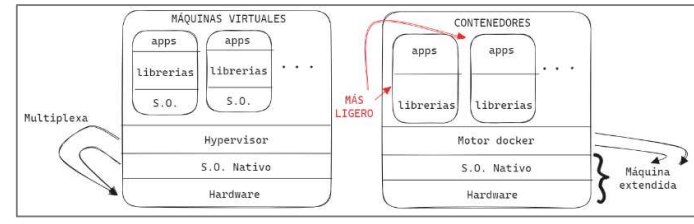
Service Level Agreement (*SLA*): Todo servicio establece un SLA.

- **Función**: Define que hace. • **Rendimiento**: Que capacidad y time respuesta. • **Disponibilidad**: % Tiempo activo.

Desplegar un servicio = instalación, activación, actualización y adaptación del servicio

- **Instalación**: Resolver **dependencias**, **configurar** software, **determinar número de instancias** de cada componente y como se reparten, **establecer orden** en que arranca.
- **Reactivación**: **Detener el sistema** de forma ordenada.
- **Actualización**: **Reemplazar/Mejorar componentes**.
- **Adaptación**: Seguir proporcionando servicio mientras: se actualiza falla, cambia, escala y reparto carga…

Docker: **Definen requisitos que hace falta para ejecutar un componente** (*worker o bróker), tanto el sistema operativo, que use una intel-9, cierta RAM…* Todo eso se puede definir en un **Docker/contenedor** e instalar este en cualquier nodo que vaya a ejecutar mi servicio, de manera que no te preocupas por DONDE lo instalas. Ya se apaña él para simular y ejecutar mi código en ese nodo concreto aplicando la conf necesaria.



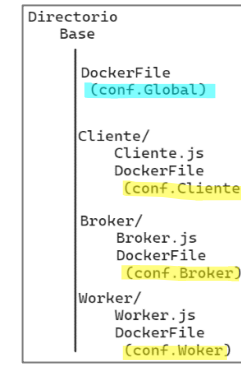
Virtualización: Te permite **mediante dockers** (*o máquinas virtuales*) instalar algo en un hardware específico para que se comporte de la manera deseada. *Lo que te haga falta, simular RAM, sistemas operativos… lo instalas y simula*.

Adaptable: Puedes **cambiar la cantidad de nodos** sin que afecte al servicio.

Elasticidad: Capacidad de poder cambiar la cantidad de wokers/brokers/nodos que crean el servicio **según la carga de manera automática**.

AUTOMATIZACIÓN DEL DESPLIEGUE

Despliegue a gran escala *no posible a mano* → Necesitamos **automatización** (*herramienta*).

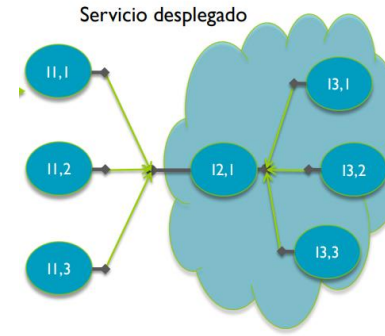


Configurar cada componente (*NO INSTANCIA*): **Defines para cada cosa** (*worker, bróker…*) **qué le hace falta**: ***Dependencias***, *sistema operativo, ram, códigos, ficheros…* Después, la herramienta generará una configuración especifica en base a la configuración para cada instancia (nodo) en la que se lance ese componente.

Plan de configuración global: Define como se conectan entre ellos los componentes, donde se coloca cada instancia (*cada componente en qué nodo*) los **Enlaces** ('bindings') **de dependencias** (*empareja endpoints, incluyendo dependencias con servicios externos*).

Ejemplo

Si voy a tener el patron Cliente-bróker-worker, mi plan sería: Arrancar Broker, Workers y luego clientes. Crear los endpoints de frontend y backend en el bróker y como no habrá dependencia externas me da igual (no otros servicios)



Tendré una instancia de Broker que al lanzarla le doy puerto del backend y el front end. También me quedo con su ip.

Los clientes y los workers necesitan saber id y ports de bróker, por lo que tienen dependencias con él. Por lo tanto, al lanzarlos les daré como argumento la ip y el puerto que les toque.

Para que todo esto funcione se tendrás plantillas de configuración para cada componente para que haga esto.

Resolver dependencias

Código resuelve las dependencias: Se leen los datos de un ficho, recibiendo datos de un socket…

Inyección de dependencias: Es lo que de normal usamos. El código de la aplicación **expone nombres locales para sus interfaces relevantes** y El **contenedor** **rellena las variables con instancias**, le pasa argumentos necesarios.

DESPLIEGUE EN LA NUBE

Se han de haber creado y seleccionado ya los componentes para el servicio. Ahora **falta seleccionar un proveedor**:

IaaS: Se nos facilita la infraestructura **SOLO EL HARDWARE**. Se basa en la virtualización usando máquinas virtuales. Es mucho más flexible en la asignación de recursos. Desventajas:

- De cara a **automatizar el despliegue es muy pobre**, *no es fácil xq tenemos que especificar al 100% todo*.
- Tampoco permite elegir características de la red **ni tenemos ayuda** en caso de que se produzcan fallos.

PaaS: Se nos **proporciona una plataforma que trata de automatizar todo lo relacionado con el despliegue**. SE CENTRA EN EL SLA de todos los componentes. El proveedor se apaña para configurar TODO según el SLA que le digamos (*automatizado y configuración ya solucionados*)

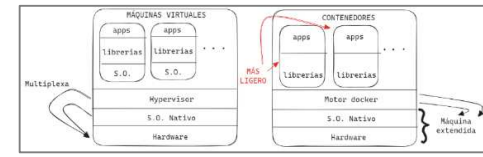
- Además, **se establecen de manera automática los nodos necesarios** para suplir la demanda.

CONTENEDORES

Los contenedores simplifican la resolución de dependencias. Crea un enlace entre las dependencias y los endpoints de manera que cada componente se puede conectar fácilmente donde toca.

Aprovisionamiento: **Reservar la infraestructura necesaria** para una aplicación distribuida. Recursos para interconectar instancias y recursos para cada instancia en concreto. Para esto hay 2 opciones

- **Máquinas virtuales**: Son un **SO** (diferente de la máquina donde se instalan) y las **bibliotecas**. ***Chupan muchos recursos***.
- **Contenedores**: Solo las **bibliotecas**. Se ejecuta sobre una máquina con un "docker engine" que **permite al contenedor usar el SO del host** para funcionar perfectemente. Es mucho **más ligero** y se puede tener muchas instacias.



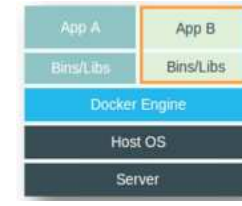
Inconvenientes: Tenemos menos flexibilidad xq la aplicación tiene que funcionar (*compatible*) en el SO del host.

Ventajas: Más ligero ya que ahorramos en el SO, más fácil de desplegar (tiempo y dificultad), bueno en dif casos.

DOCKER

El **fichero de configuración Dockerfile** automatiza el despliegue de cada instancia (Configurar, encender, parar, actualizar…). Además, define un sistema ficheros para compartir cosas entre contenedores ahorrando espacio.

Componentes



- **Imagen**: **Plantilla** de solo lectura con el **Conjunto de instrucciones** (ordenes) **para crear un contexto de ejecución**: Instalar el programa y las bibliotecas para ejecutarse.
  $$Imagen\ nueva\ =\ Imagen\ base\ +\ instrucciones.$$
  - **Imágenes base**: hay imágenes ya hechas de las que derivan las demás.
- **Contenedor**: Son los **recursos** que necesita una instancia para ejecutarse, recursos especificados en la imagen: Es una **imagen ya creada en ejecución**. *Maquina independiente con determinado contexto. Este contexto son los ejecutables, las bibliotecas y aplicaciones... Este contexto consume recursos*.
- **Depósito**: Hay un **depósito local** (donde están las imágenes que puedes ejecutar) y uno en la nube en la que hay imágenes que me puedo descargas y usar. También las puedes crear y subir para que otros las descarguen.

A esos depósitos ya hechos se les pueden añadir instrucciones o lo necesario para instalar lo que se requiera y poder personalizarlos. *NO los creas desde 0, partes de una base ya hecha y la customizas, creas extensiones*.

Las imágenes base no son como tal nada, ni un archivo ni nada. Cuando las configurar por alguna de las 2 rutas siguientes es cuando tienes tu imagen. Puedes ejecutar las veces que quieras, distribuirlas etc.

Crear contenedor

Crear e iniciar contenedor desde imagen: `docker run opciones imagen progInicial`

Opciones: -i -t parar iniciarlo en modo interactivo con la consola abierta de manera que puedes ejecutar cosas.

Ejemplo: `docker run -i -t ubuntu bash`. Descarga la imagen Ubuntu, crea el contenedor y reserva recursos y ejecuta el programa "bash".

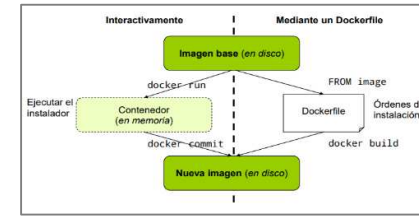
Crear nueva imagen: Con la consola podríamos modificar el contenedor y a partir de este, crear una nueva imagen para ejecutar. Para guardar los cambios: `docker commit nombreContenedor nombreImagen`.

Eso guarda todas las nuevas configuraciones y lo deja listo para ejecutarlo y tener un contenedor diferente.

CREAR NUVEAS IMÁGENES

Para crear una imagen a partir de una imagen base se usan ordenes de consola. Se deben de añadir las bibliotecas, el interprete y el programa a ejecutar. Hay 2 formas de crear una nueva imagen: **Interactivamente** y con un **Dockerfile**. **Ambas parten de una imagen base**.

Interactivamente: Ejecutar la imagen con `Docker run` y hacer que se pueda interactuar con la línea de comandos para ejecutar/instalar y configurar lo que queramos.



Cuando hayamos terminado para "exportar la imagen" hacemos `docker ID/NombreContenedor commit nombredestino`.

Dockerfile: Un fichero de configuración. En la primera línea se especifica QUE IMAGEN base se va a modificar. En el archivo se **escriben las MISMAS líenas que se ejecutarían en el bash a mano**. Cuando el archivo esté ready: "`Docker build -t tsr-zqm .`"

Una vez tengas las imágenes creadas y subidas, si tienes un cluster o máquinas diferentes que tengan todas un motor Docker, puedes iniciar en todas a la vez la imagen y hacer que ejecuten lo que quieras. DIAPOS 38 y 39.

Id del contenedor: Cuando tenemos un contenedor "ejecutado" con `docker ps -a` podemos ver su id

Ordenes en el fichero Dockerfile: Cada instrucción empieza por una orden en mayúsculas

FROM: Sirve para **elegir de qué imagen se va a partir**, modificar una imagen base. SE PONE AL PRINCIPIO

RUN orden: Ejecuta dicha orden en el Shell.

ADD origen destino: Copia fecheros del path origen al path destino. Si origen es directorio lo copia entero, si es un comprimido lo extrae. **COPY origen destino**: Es igual que ADD, pero no expande los ficheros comprimidos

EXPOSE puerto: **Indica el puerto** en el que el contenedor atenderá peticiones.

WORKDIR path: Indica el **directorio de trabajo** para las órdenes RUN, CMD, ENTRYPOINT.

ENV variable valor: **Asigna valor a una variable de entorno** accesible por los programas. RESOLV DEPENDENCIA.

CMD orden arg1 arg2 ... Proporciona valores por defecto para la **ejecutar ordenes en del contenedor**.

- **ENTRYPOINT** orden arg1 arg2 ... Ejecuta la orden **al crear el contenedor** (***termina al finalizar la orden***).

Solo debería haber una orden CMD o ENTRYPOINT, sirven para lanzar el programa: CMD node worker.js 9000…

Crear: Al acabar hay que hacer `Docker build -t nombre .`. el punto es el directorio donde está.

MULTIPLES COMPONENTES

Múltiples componentes en un mismo nodo

Si queremos lanzar varias componentes como c-b-w, hace falta que se **lanza el bróker antes de crear las imágenes** del worker y los clientes. Cuando se haya lanzado, se le pillan el puerto y la ip donde escucha, se modifica el Dockerfile de workers y clientes, y se buildean y lanzan.

Esto es costoso y tedioso, xq además un componente se tiene que ejecutar ANTES de crear las imágenes.

Sería mejor **automatizar esto**, crear un plan de trabajo con la descripción de los compontes y su relación entre ellas, de manera que las dependencias se solucionen dinámicamente en tiempo de ejecución. -> Docker compose.

Docker-compose: *Múltiples componentes de manera manual, dependencias*

Se han de crear Dockerfiles que sean configurables: Al lanzar las componentes, en vez de darle un valor a cada argumento, **le pasas una variables de entorno** a la que le darás valor cuando se resuelva la dependencia.

BROKER

```
FROM tsr2021/ubuntu-zmq
COPY ./broker.js broker.js
EXPOSE 9998 9999
CMD node broker
```

WORKER

```
FROM tsr2021/ubuntu-zmq
COPY ./worker.js worker.js
CMD node worker $BROKER_URL
```

CLIENTE

```
FROM tsr2021/ubuntu-zmq
COPY ./client.js client.js
CMD node client $BROKER_URL
```

Después de crear los docker files, usas la herramienta Docker-compose para lanzarlo todo y que se asignen bien los valores de las variabes. AQUÍ ESTARÍA EL PLAN DE DESPLIGUE.

```
version: '2'
services:
    cli:
        image: client
        build: ./client/
        links:
            - bro
        environment:
            - BROKER_URL=tcp://bro:9998
    bro:
        image: broker
        build: ./broker/
        expose:
            - "9998"
            - "9999"
    wor:
        image: worker
        build: ./worker/
        links:
            - bro
        environment:
            - BROKER_URL=tcp://bro:9999
```

Nombre y construcción: Se especifica **PARA CADA Componente** qué **nombre tiene** y **DONDE está el Dockerfile** a ejecutar SI NO EXISTE ESA IMAGEN.

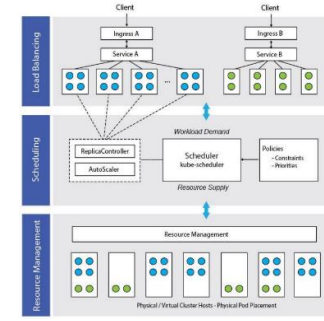
Conexiones y dependencias: El broker **EXPONE** los puertos en los que escuchará peticiones. Los demás con "`link`" dicen que quieren algo de ese componente para **ASIGNAR VALOR A LAS VARIABLES DECLARADAS ANTES**.

Ejecutar: se lanza con `docker-compose up -d`. Si NO ESTÁN, se creen las imágenes de los componentes y arrancar las instancias de cada componente.

- Lanzar **múltiples instancias de un componente**: `docker-compose up -d` **`--scale X=n`**. X es el nombre la componente: "cli", "wor"…

Múltiples componentes en distintos nodos

Para lanzar las componentes en diferentes nodos se pueden usar PaaS o clusters. Docker-compose se limita a organizar componentes en un único nodo. **Kubernetes Nos permiten distribuir las instancias** entre los distintos nodos, es un orquestador/distribuidor de contenedores (pero nada que ver con Docker).

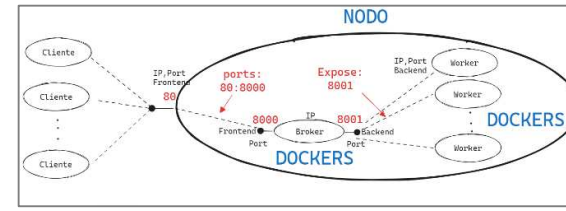


Kubernetes: Se compone de nodos (clusters) tanto virtuales como físicos. También tiene pequeñas unidades "pods" para despliegue. Y otras cosas que paso de escribir

Fuck esta parte del tema, si preguntan algo me corto los huevos.

Ejercicios y cosas a tener en cuenta

Expose port: Las componentes, aunque se lancen en el mismo nodo, **NO COMPARTEN PUERTOS**. Esto quiere decir que pueden hacer "expose 8000" y cada una atiende en su puerto personal 8000. NO HABRÍA COMFLICTO.



Ports port: Expose dice DONDE, en qué puerto, va a escuchar cada contenedor. **Ports** lo que hace es, de los **puertos en los que yo escucho**, los **conecto a un puerto real de la máquina**. NO se puede hacer Ports 2 veces en el mismo puerto xq ahí SÍ que es SOLO 1 puerto. Por lo que daría fallo.