

Tema 4. Despliegue de Servicios. Docker

Tecnologías de los Sistemas de Información en la Red



Índice

1. **Objetivos**
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias



I. Objetivos

- ▶ Introducir el concepto de despliegue de una aplicación distribuida.
- ▶ Analizar los problemas derivados de la existencia de dependencias.
- ▶ Discutir cómo tratar el despliegue y sus problemas en un sistema genérico.
- ▶ Proporcionar herramientas para facilitar el despliegue.
- ▶ Aplicación a un caso concreto.



Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias



2. Concepto de despliegue

- ▶ **Despliegue:** Actividades que hacen que un sistema software esté preparado para su uso.
- ➡ Actividades relacionadas con la *instalación, activación, actualización y eliminación* de componentes o del sistema completo.

La depuración no formaría parte del despliegue como tal, sería como el punto final antes de iniciar el despliegue.



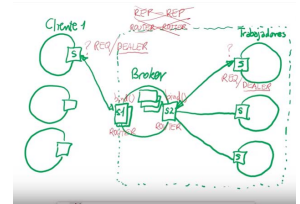
2.1. Despliegue de una aplicación distribuida

- ▶ Una aplicación distribuida es una colección de componentes heterogéneos dispersos sobre una red de computadores
 - ▶ Muchos componentes, creados por desarrolladores distintos
 - ▶ Pueden cambiar de forma rápida e independiente (p.ej., nuevas versiones, etc.)
- ▶ Los componentes de la aplicación deben cooperar → existen dependencias entre ellos
- ▶ Los nodos pueden ser heterogéneos (distintos equipos, sistemas operativos, etc.), pero cada componente tiene ciertos requisitos para su ejecución.
- ▶ Podemos tener requisitos adicionales de seguridad (privacidad, autenticación, etc.)

2.2. Ejemplo de despliegue

Suponemos el patrón *broker* desarrollado en la práctica 2

- ▶ Formado por 3 componentes (*client*, *broker*, *worker*) básicamente autónomos
 - ▶ Podrían estar desarrollados en lenguajes distintos, por programadores diferentes
- ▶ Número variable de instancias de cada componente (ej. varios clientes, o varios trabajadores). Cada instancia:
 - ▶ Puede iniciarse/detenerse/reiniciarse con independencia del resto de instancias
 - ▶ Falla de forma independiente del resto
 - ▶ Tiene una ubicación propia (independiente del resto)





2.2. Ejemplo de despliegue: dependencias y requisitos

- ▶ Un cliente
 - ▶ Debe conocer la ubicación del *broker* (IP y puerto *frontend*)
 - ▶ Debe poseer una identidad única -> se resuelve utilizando un *router*
- ▶ Un trabajador
 - ▶ Debe conocer la ubicación del *broker* (IP y puerto *backend*)
 - ▶ Debe poseer una identidad única -> se resuelve utilizando un *router* o, *dealer* con reparto circular (no tan recomendable)
- ▶ Todos los componentes requieren determinado entorno de ejecución (NodeJS y ZeroMQ)



2.2. Ejemplo de despliegue: despliegue manual

- ▶ Copiar el código fuente de cada componente en aquellas máquinas donde debe ejecutarse una instancia
 - ▶ Debemos garantizar que en cada uno de esos nodos está correctamente instalado el software base (NodeJS y ZeroMQ) con las versiones correctas
- ▶ Lanzar las instancias de los distintos componentes en el orden correcto (*broker*, trabajadores, clientes)
 - ▶ En la línea de órdenes usada para lanzar cada instancia, indicar los argumentos necesarios (ej. IP y puerto para conectar con el *broker*, identidad, etc.)

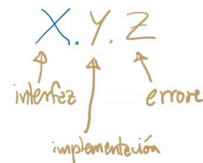


Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias

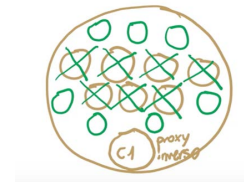
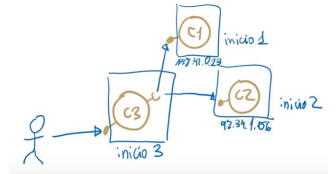
3. Despliegue de un servicio

- ▶ Desarrollamos sistemas distribuidos para ofrecer **servicios** (funcionalidad) a clientes remotos
 - ▶ Aplicación + Despliegue = Servicio
- ▶ Todo servicio establece un SLA (*Service Level Agreement*)
 - ▶ Definición funcional (qué hace)
 - ▶ Rendimiento (qué capacidad tiene, tiempos esperados de respuesta, ...)
 - ▶ Disponibilidad (porcentaje de tiempo en que el acceso al servicio está garantizado)
 - ▶ Aunque existen servicios efímeros (disponibles durante cortos periodos de tiempo), nos centramos en los persistentes (disponibilidad continua) Ej. Gmail, Dropbox, ...
- ▶ **Desplegar un servicio** = instalación, activación, actualización y adaptación del servicio

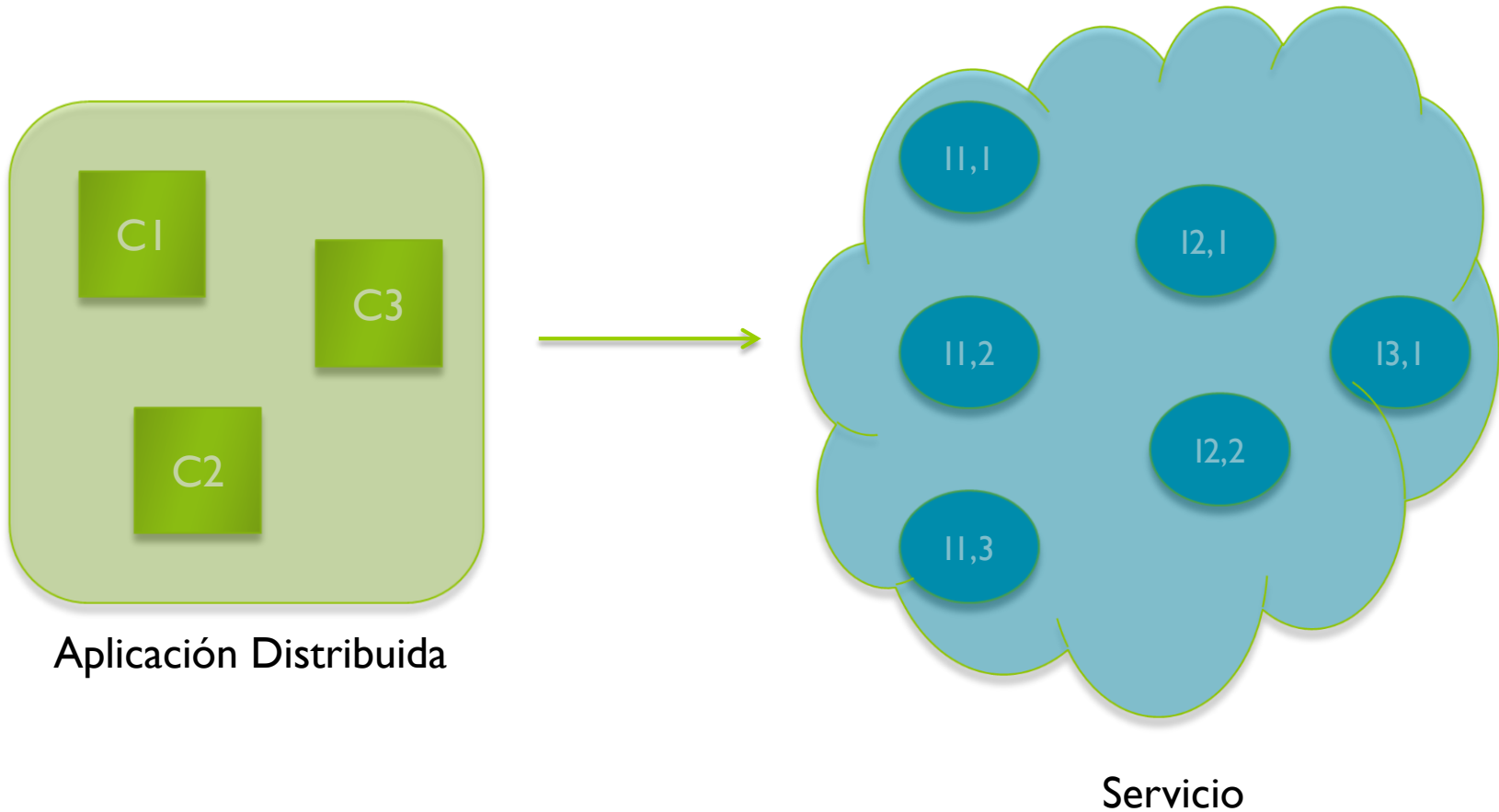


3. Despliegue de un servicio

- ▶ Instalación y activación.- ejecución del *software*
 - ▶ Resolver las dependencias del *software* (ej. bibliotecas, etc.)
 - ▶ Configurar el *software* (versiones compatibles, etc.)
 - ▶ Determinar el número de instancias de cada componente y su reparto entre los distintos nodos
 - ▶ Resolver las dependencias entre agentes (ej. puertos)
 - ▶ Establecer el orden en que arrancan los componentes
- ▶ Desactivación.- detener el sistema de forma ordenada
- ▶ Actualización.- reemplazar componentes (ej. nueva versión)
- ▶ Adaptación (sin detener el servicio) tras:
 - ▶ Fallo/recuperación de un agente
 - ▶ Cambios en la configuración de los agentes
 - ▶ Escalado (reacción ante cambios en la carga)



3. Despliegue de un servicio





Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias



4. Automatización del despliegue

Un despliegue a gran escala no puede hacerse a mano → necesitamos automatización (herramienta)

- ▶ Configuración para cada componente
 - ▶ Fichero con lista de parámetros de configuración y descripciones de dependencias
 - ▶ La herramienta genera una configuración específica para cada instancia de dicho componente
- ▶ Plan de configuración global
 - ▶ Plan de conexión entre componentes (lista de *endpoints* expuestos, lista de dependencias)
 - ▶ Decide dónde colocar cada instancia
 - ▶ Enlace ('*binding*') de dependencias (empareja *endpoints*, incluyendo dependencias con servicios externos)

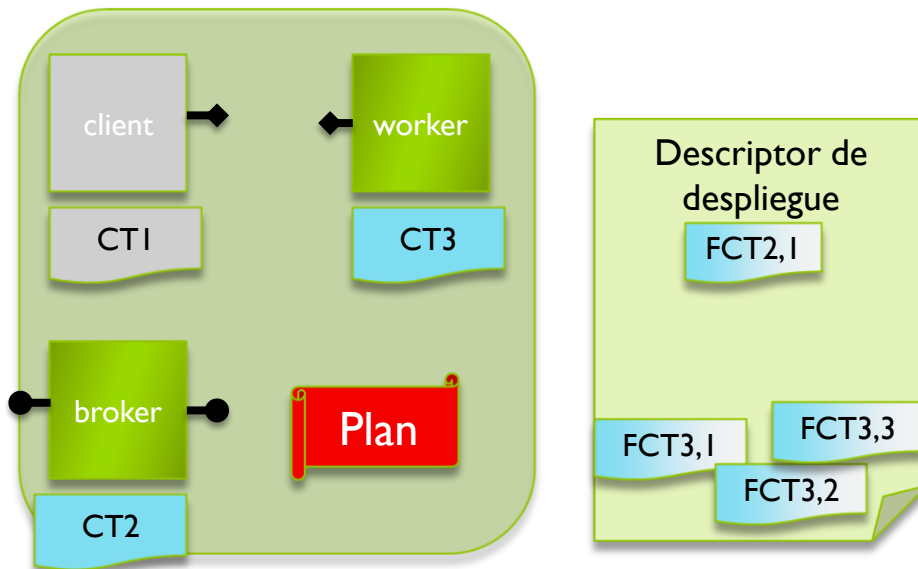


4. Automatización: Ejemplo

- ▶ Varias instancias de *client*
 - ▶ Argumentos frontendURL, id
 - ▶ Dependencia respecto a *broker*
- ▶ Una instancia de *broker*
 - ▶ Argumentos frontendPort, backendPort
- ▶ Varias instancias de *worker*
 - ▶ Argumentos backendURL, id
 - ▶ Dependencia respecto a *broker*
- ▶ Plan
 - ▶ El orden de arranque es *broker, workers, clients*
 - ▶ Los endpoints son el frontend (externo) y el backend (interno)
 - ▶ No habrá dependencias respecto a servicios externos

4. Automatización: Ejemplo

Aplicación distribuida



◆ Dependencia

● Endpoint

◆ Dependencia resuelta

● Endpoint resuelto

● Endpoint expuesto del servicio



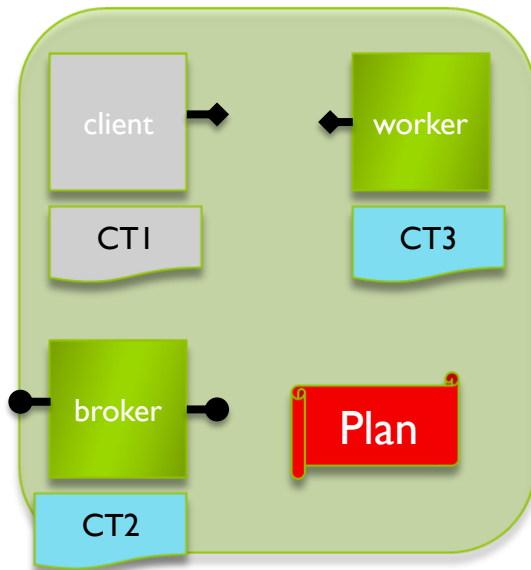
Plantilla de configuración del componente i



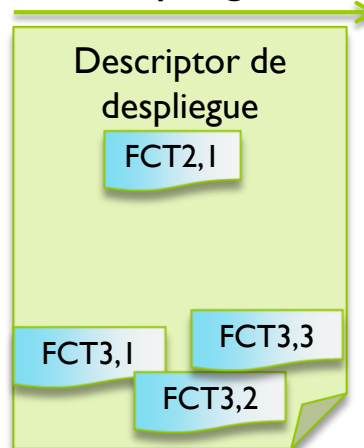
Plantilla de configuración cumplimentada para la instancia j del componente i

4. Automatización: Ejemplo

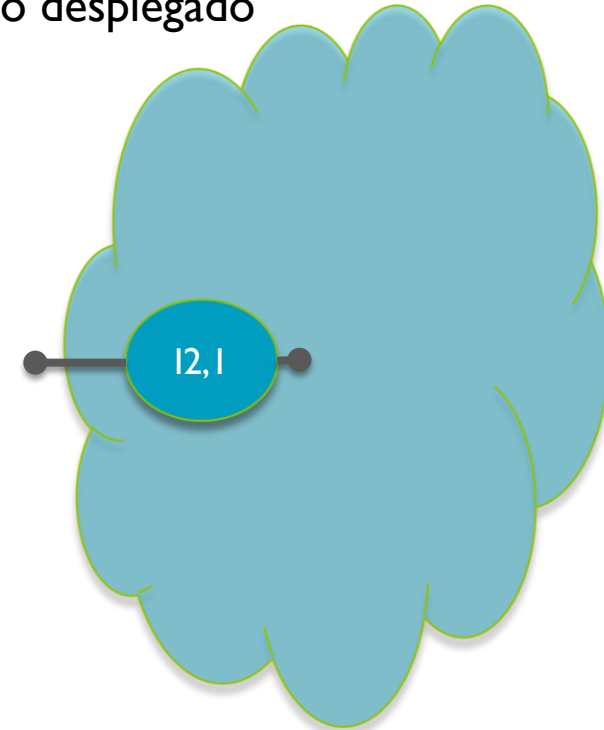
Aplicación distribuida



Despliegue



Servicio desplegado



◆ Dependencia

● Endpoint

◆ Dependencia resuelta

● Endpoint resuelto

● Endpoint expuesto del servicio



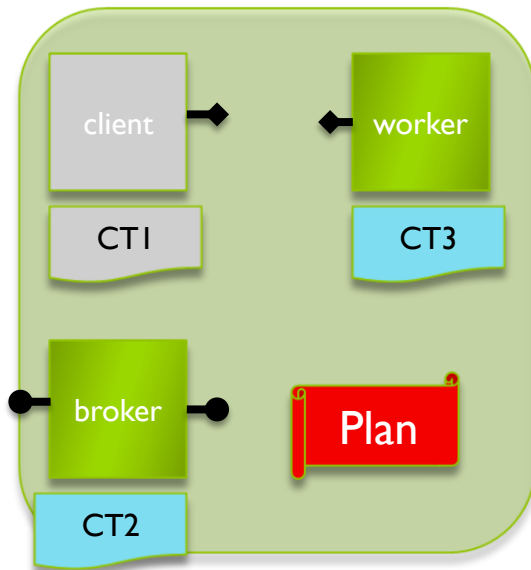
Plantilla de configuración del componente i



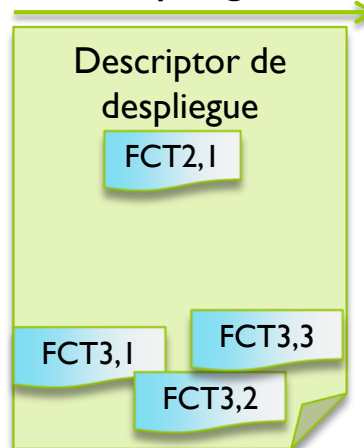
Plantilla de configuración cumplimentada para la instancia j del componente i

4. Automatización: Ejemplo

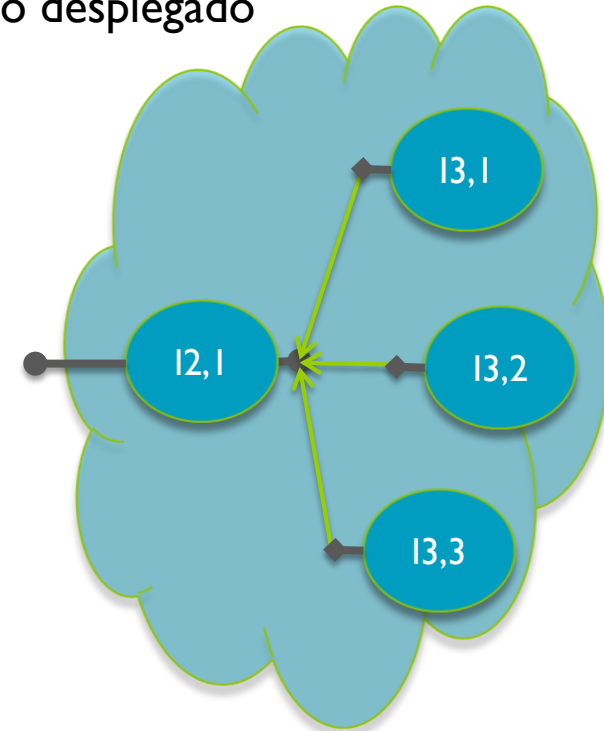
Aplicación distribuida



Despliegue



Servicio desplegado



◆ Dependencia

● Endpoint

◆ Dependencia resuelta

● Endpoint resuelto

● Endpoint expuesto del servicio



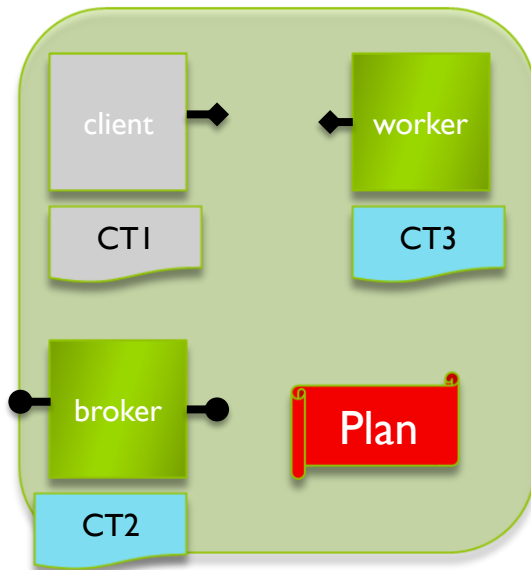
Plantilla de configuración del componente i



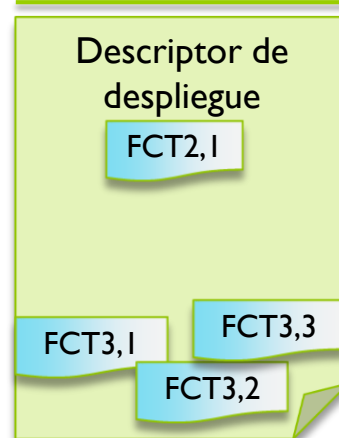
Plantilla de configuración cumplimentada para la instancia j del componente i

4. Automatización: Ejemplo

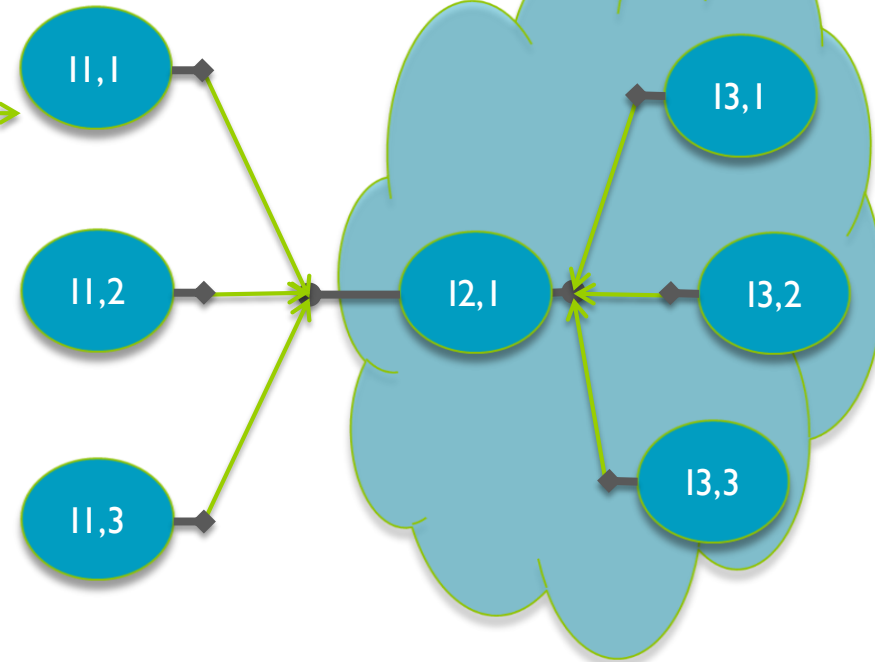
Aplicación distribuida



Despliegue



Servicio desplegado



◆ Dependencia

● Endpoint

◆ Dependencia resuelta

● Endpoint resuelto

● Endpoint expuesto del servicio



Plantilla de configuración del componente i



Plantilla de configuración cumplimentada para la instancia j del componente i



4. Automatización del despliegue

Resolución de dependencias. Opciones:

1. El código define la forma de resolver las dependencias
 - ▶ Ej. leyendo datos de un fichero, o recibiendo datos en un socket
 - ▶ Bajo nivel
2. **Inyección de dependencias** (recomendado)
 - ▶ El código de la aplicación expone nombres locales para sus interfaces relevantes
 - ▶ El contenedor rellena las variables con instancias de objetos
 - ▶ Crea un grafo de las instancias de los componentes del servicio
 - ▶ Los arcos del grafo son enlaces dependencia-*endpoint*



Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias



5.1. Despliegue en la nube: IaaS

- ▶ Se basa en virtualización
 - ▶ Máquinas virtuales de distintos tamaños
 - ▶ Flexibilidad en la asignación de recursos
- ▶ Presenta limitaciones en el despliegue
 - ▶ Decisiones de asignación no automáticas (bajo nivel)
 - ▶ Número de instancias por componente, ubicación, tipo de MV
 - ▶ No permite elegir características red (retardo, ancho de banda)
 - ▶ Modelo de fallo insuficiente
 - ▶ Los modos de fallo no son realmente independientes
 - ▶ Ayuda limitada a la recuperación



5.2. Despliegue en la nube: PaaS

- ▶ SLA como elemento central → parámetros del SLA para todos los componentes
 - ▶ Se persigue la automatización del despliegue
 - ▶ Planes de despliegue a partir del SLA (*Service Level Agreement*)
 - ▶ Planes para actualización/configuración
 - ▶ Situación actual
 - ▶ El grado de automatización ha mejorado en los últimos años
 - Cubre el despliegue inicial y el autoescalado de servicios sencillos
 - Todavía no garantiza que la actualización de los servicios respete su SLA
 - ▶ Microsoft Azure es uno de los más evolucionados

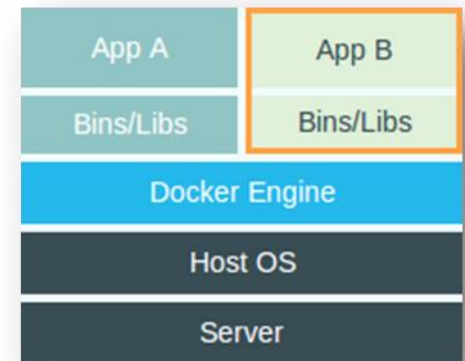
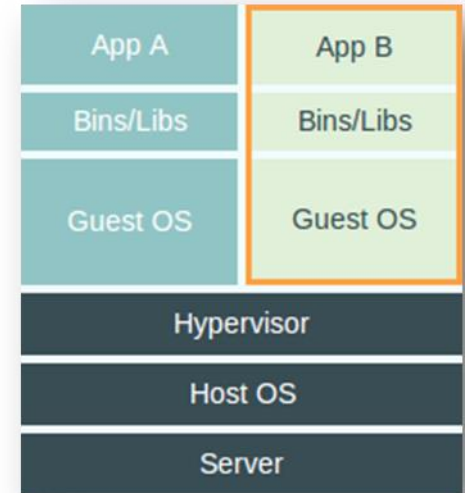


Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias

6. Contenedores

- ▶ **Aprovisionamiento** = reservar la infraestructura necesaria para una aplicación distribuida
 - ▶ Recursos para intercomunicación entre instancias
 - ▶ Recursos para cada instancia (procesador, memoria, ...). Alternativas:
 - ▶ Instancia sobre **MV** → SO + Bibliotecas
 - ▶ Instancia sobre **contenedor** (versión ligera de la MV) → Bibliotecas
 - Usa el SO del anfitrión



6. Contenedores

Suponemos el uso de contenedores en lugar de MV

- ▶ Menor flexibilidad
 - ▶ El *software* de la instancia ha de ser compatible con SO anfitrión
 - ▶ El aislamiento entre contenedores no es perfecto
- ▶ Utiliza muchos menos recursos
 - ▶ Ej.: desplegamos 100 instancias de un componente cuya ejecución requiere 900 MB (SO) + 100 MB (resto)
 - ▶ con MV: $100 * (900 \text{ MB} + 100 \text{ MB}) = 100 \text{ GB}$
 - ▶ con contenedores: $900 \text{ MB} + 100 * 100 \text{ MB} = 10.9 \text{ GB}$
 - ▶ Ahorramos espacio y tiempo (ej. para instalar la imagen)
- ▶ Mayor facilidad de despliegue (fichero de configuración)
- ▶ Aplicable en casi todos los escenarios



6. Contenedores: Docker

- ▶ El fichero de configuración Dockerfile automatiza el despliegue de cada instancia
- ▶ Soporta control de versiones (Git)
- ▶ Además del sistema de ficheros nativo, define un sistema ficheros de solo lectura para compartición entre contenedores
- ▶ Permite cooperación en el desarrollo mediante depósitos públicos

NOTA 1.- Aunque existe Docker para Windows, asumimos que el S.O. del huésped es Linux

NOTA 2.- Para el apartado próximo (Docker) es necesario consultar la *Guía del Alumno* y la web **www.docker.com**



Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias



7. Docker: Componentes

1. **Imagen:** plantilla de solo lectura con las instrucciones para crear un contenedor
 - ▶ Ej.- podemos crear una imagen para proporcionar Linux+NodeJS+ZeroMQ, a la que denominamos ubuntu-zmq
2. **Contenedor:** conjunto de recursos que necesita una instancia para ejecutarse. Se crea al ejecutar una imagen
 - ▶ Ej.- para probar el código de la práctica 2, ejecutamos cada instancia sobre un contenedor creado a partir de ubuntu-zmq
3. **Depósito:** lugar donde podemos dejar/obtener imágenes (espacio para compartir imágenes)
 - ▶ Ej.- Podemos subir la imagen ubuntu-zmq, y cualquiera puede bajarla y usarla para crear contenedores



7.1. Docker: Imagen

- ▶ En el depósito existen imágenes predefinidas para las distintas distribuciones Linux (ej. imagen `ubuntu:focal`)
- ▶ Nueva imagen = imagenBase + instrucciones. Ejemplo:
 - ▶ `ubuntu:focal` + instr. para instalar node y zmq → `ubuntu-zmq`
 - ▶ `ubuntu-zmq` + ... → creamos imágenes para cada componente (*client, broker, worker*)
- ▶ Docker utiliza órdenes desde consola
- ▶ Estructura general: **docker acción opciones argumentos**
 - ▶ Información sobre las imágenes a nivel local: `docker images`
 - ▶ Información sobre una imagen: `docker history nombreImag`



7.2. Docker: Contenedor

- ▶ Crear e iniciar contenedor desde imagen

```
docker run opciones imagen programaInicial
```

- ▶ Ej. `docker run -i -t ubuntu bash` descarga la imagen ubuntu, crea el contenedor, reserva sistema de ficheros, reserva interfaz de red y dirección IP interna, y ejecuta bash
 - ▶ Las opciones `-i -t` indican modo interactivo (la consola queda abierta y conectada al contenedor)
- ▶ Modificamos el contenedor mediante órdenes en modo interactivo desde consola
- ▶ Crear nueva imagen a partir del estado actual del contenedor

```
docker commit nombreContenedor nombreImagen
```




7.2. Docker: Grupos de operaciones

Grupo	Descripción
config	Gestión de configuraciones
container	Operaciones sobre contenedores
context	Contextos para el despliegue distribuido (k8s, ...)
image	Gestión de imágenes
network	Gestión de redes
service	Gestión de servicios (contenedores instanciados desde la misma imagen) como parte de una aplicación distribuida (p.ej un SGBD)
system	Gestión global de Docker
volume	Gestión de almacenamiento persistente



7.2. Docker: Algunas órdenes destacables

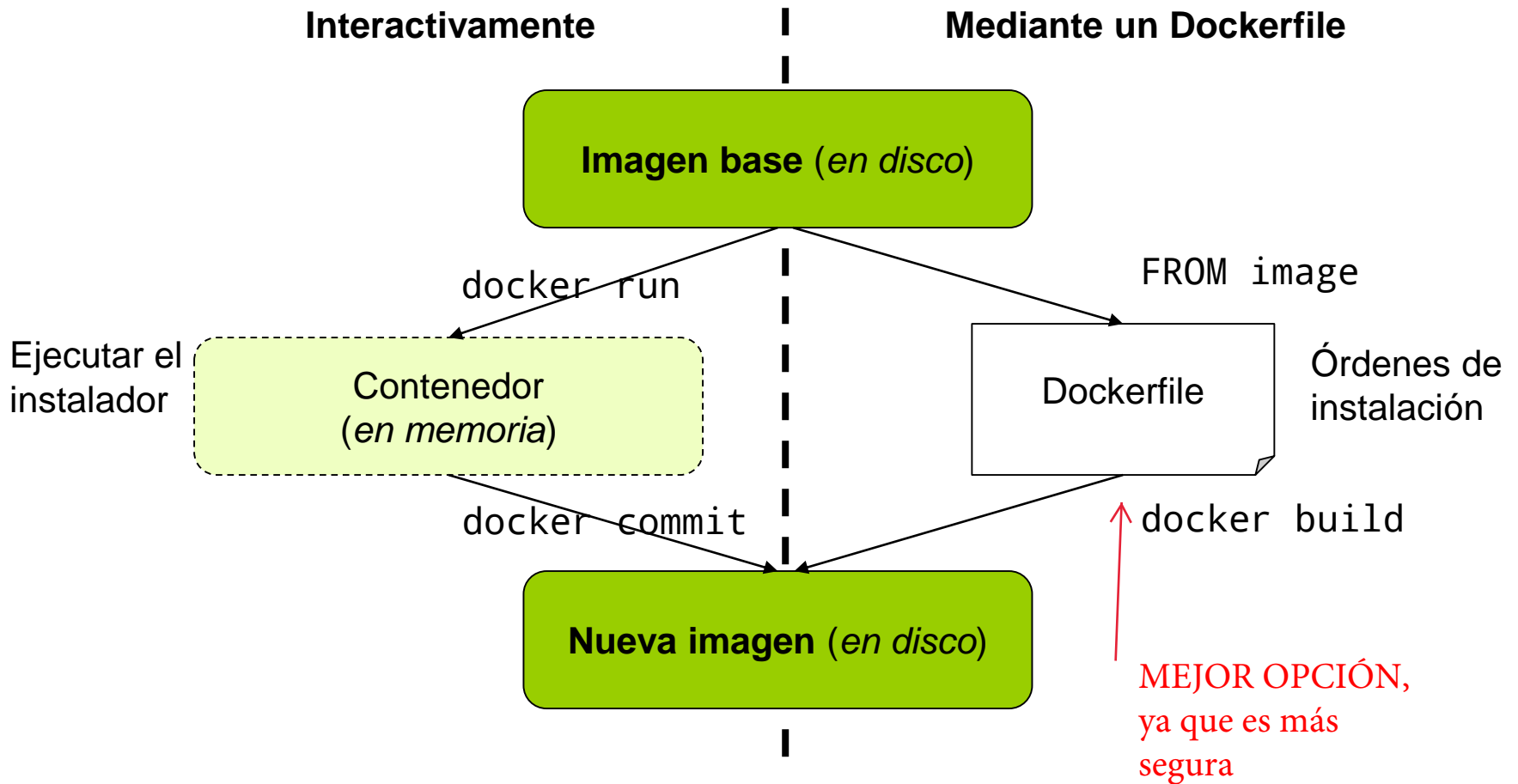
1. Control del ciclo de vida (grupo container)
 - ▶ **docker commit | run | start | stop**
2. Informativas (diversos grupos)
 - ▶ **docker logs | ps | info | images | history**
3. Acceso al depósito (grupo image)
 - ▶ **docker pull | push**
4. Varias (grupo container)
 - ▶ **docker cp | export**



Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias

8. Creación de una imagen





8. Creación de una imagen

- ▶ Dos alternativas para crear una imagen:
- ▶ Interactivamente
 - ▶ Tomar una imagen base y crear el contenedor:
`docker run -i -t imagen progInicial`
 - ▶ Modificar interactivamente el contenedor
 - ▶ Guardar ese estado como una nueva imagen:
`docker commit idContenedor nuevaImagen`
- ▶ Crear una nueva imagen a partir de las instrucciones de un fichero de texto denominado Dockerfile
`docker build -t nombreNuevaImagen pathDockerfile`



8.1. Creación interactiva de imagen. Ejemplo

- ▶ Imagen que permite ejecutar programas NodeJS usando 'zeromq'.

- Paso 1: Lanzar Docker seleccionando una imagen Ubuntu interactiva que ejecute el shell:

```
$ docker run -i -t ubuntu:focal bash
```

- Paso 2: Lanzamos las siguientes órdenes en el contenedor:

```
$ apt-get update
$ apt-get -y install curl ufw gcc g++ make gnupg
$ curl -sL https://deb.nodesource.com/setup_12.x | bash -
$ apt-get -y install nodejs
$ apt-get -y upgrade
$ npm install zeromq@5
$ exit -> se para el contenedor.
```

- Desde línea de órdenes del anfitrión, obtenemos nombre e ID del contenedor: `docker ps -a`
 - Creamos la nueva imagen:
 - `docker commit IDoNombreContenedor nombreNuevaImagen`



8.2. Creación de imagen con Dockerfile. Ejemplo

1. Escribimos el fichero de texto llamado Dockerfile:

```
FROM ubuntu:focal
RUN apt-get update
RUN apt-get install -y curl ufw gcc g++ make gnupg
RUN curl -sL https://deb.nodesource.com/setup_12.x | bash -
RUN apt-get -y install nodejs
RUN apt-get -y upgrade
RUN npm install zeromq@5
```

2. Nos situamos en el directorio donde está ese Dockerfile

3. Ejecutamos esta orden:

▶ `docker build -t tsr2021/ubuntu-zmq .`



8.3. Docker. Fichero Dockerfile

- ▶ Cada línea empieza con una instrucción en mayúsculas
- ▶ La primera instrucción (primera línea) debe ser **FROM *imagenBase***
- ▶ **RUN *orden*** ejecuta dicha orden en el shell
- ▶ **ADD *origen destino***
 - ▶ Copia ficheros de un lugar (URL, directorio, o archivo) a un path en el contenedor
 - ▶ Si el origen es un directorio, lo copia completo. Si es un fichero comprimido, lo expande al copiar
- ▶ **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



8.3. Docker. Fichero Dockerfile

- ▶ **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 dentro del contenedor
- ▶ **CMD** *orden arg1 arg2 ...* proporciona valores por defecto para la ejecución del contenedor
- ▶ **ENTRYPOINT** *orden arg1 arg2 ...* ejecuta dicha orden al crear el contenedor (termina al finalizar la orden)

Sólo debería haber como máximo una orden CMD o ENTRYPOINT (si hay mas, sólo ejecuta la última)



Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias

9. Múltiples componentes en un nodo. Ejemplo

- ▶ Ejemplo: Servicio implantado mediante un programa “broker” y otro programa “worker”, que podrá replicarse tantas veces como sea necesario.
 - ▶ Prácticamente idéntico al del apartado 6.3 de la práctica 2

```
// ROUTER-ROUTER request-reply broker in NodeJS
```

```
const zmq = require('zeromq')  
let cli=[], req=[], workers=[]
```

```
let args = process.argv.slice(2)  
let fePortNbr = args[0] || 9998  
let bePortNbr = args[1] || 9999
```

```
let sc = zmq.socket('router') // frontend  
let sw = zmq.socket('router') // backend
```

```
sc.bind('tcp://*:'+fePortNbr)  
sw.bind('tcp://*:'+bePortNbr)
```

```
sc.on('message',(c,sep,m)=> {  
  if (workers.length==0) {  
    cli.push(c); req.push(m)  
  } else {  
    sw.send([workers.shift(),' ',c,' ',m])  
  }  
})
```

```
sw.on('message',(w,sep,c,sep2,r)=> {  
  if (c!='') sc.send([c,' ',r])  
  if (cli.length>0) {  
    sw.send([w,' ',  
      cli.shift(),' ',req.shift()])  
  } else {  
    workers.push(w)  
  }  
})
```



9. Múltiples componentes en un nodo. Ejemplo

► Código del worker y del cliente.

► Idem práctica 2, con argumentos en invocación y valores por omisión

```
// worker in NodeJS with URL & id arguments  
const zmq = require('zeromq')  
let req = zmq.socket('req')
```

```
let args = process.argv.slice(2)  
let backendURL = args[0] ||  
'tcp://localhost:9999'
```

```
let myID = args[1] ||  
'WID'+parseInt(Math.random()*5000)  
let replyText = args[2] || 'resp'
```

```
req.identity = myID  
req.connect(backendURL)  
req.on('message', (c, sep, msg)=> {  
  setTimeout(()=> {  
    req.send([c, '', replyText])  
  }, 1000)  
})  
req.send(['', '', ''])
```

```
// client in NodeJS with URL & id arguments  
const zmq = require('zeromq')  
let req = zmq.socket('req')
```

```
let args = process.argv.slice(2)  
let brokerURL = args[0] ||  
'tcp://localhost:9998'
```

```
let myID = args[1] ||  
'CID'+parseInt(Math.random()*10000)  
let myMsg = args[2] || 'Hola'
```

```
req.identity = myID  
req.connect(brokerURL)  
req.on('message', (msg)=> {  
  console.log('resp: '+msg)  
  process.exit(0);  
})  
req.send(myMsg)
```

9. Múltiples componentes en un nodo. Ejemplo

- ▶ Si hay varios componentes aparecen dependencias
 - ▶ El cliente necesita conocer la URL del frontend (IP y puerto)
 - ▶ El worker necesita conocer la URL del backend (IP y puerto)
- ▶ Pero no conocemos la IP hasta lanzar el contenedor del *broker*
- ▶ Alternativa manual
 - ▶ Una vez lanzado el *broker* en su contenedor, obtenemos la IP del contenedor *con docker inspect*
 - ▶ Modificamos manualmente esos valores en los Dockerfile de clientes y trabajadores para crear correctamente sus imágenes
 - ▶ Lanzamos trabajadores y clientes
- ▶ Automatización:
 - ▶ Definimos un Plan de trabajo = Descripción de componentes, propiedades y relaciones
 - ▶ Usamos una herramienta que hace el despliegue a partir del plan de trabajo



9.1. Método manual: *Broker*

- ▶ En un directorio copiamos broker.js y este Dockerfile

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

- ▶ El frontend es el puerto 9998 y el backend el 9999.
- ▶ Ejecutamos: `docker build -t broker .`
- ▶ Lanzamos el broker: `docker run -d broker`
- ▶ Averiguamos la URL del contenedor que ejecuta el broker
 - ▶ `docker ps -a` para conocer el ID del contenedor
 - ▶ `docker inspect ID` para obtener su dirección IP

9.1. Método manual: *Worker*

- ▶ En otro directorio copiamos worker.js y el siguiente Dockerfile (asumiendo que la IP del broker es a.b.c.d):

```
FROM tsr2021/ubuntu-zmq  
COPY ./worker.js worker.js  
CMD node worker tcp://a.b.c.d:9999
```

- ▶ **¡La dirección a . b . c . d debe averiguarse después de iniciar el broker!** Podemos consultarla con `docker inspect`
- ▶ En este directorio ejecutamos `docker build -t worker .`
- ▶ Lanzamos el trabajador con `docker run -d worker`
- ▶ Podemos lanzar tantas instancias como consideremos necesario



9.1. Método manual: Cliente local

- ▶ En otro directorio copiamos client.js y el siguiente Dockerfile:

```
FROM tsr2021/ubuntu-zmq  
COPY ./client.js client.js  
CMD node client tcp://a.b.c.d:9998
```

- ▶ En este directorio ejecutamos `docker build -t client .`
- ▶ Lanzamos el cliente con `docker run -d client`
 - ▶ Podemos lanzar tantas instancias como consideremos necesario



9.1. Método manual: Cliente remoto

- ▶ Lanzamos el broker con `docker run -p 8000:9998 -d broker`
 - ▶ La opción `-p portAnfitrión:portContenedor` permite acceso al frontend desde el exterior (puerto 8000 del anfitrión)
 - ▶ La opción `-d` lanza el contenedor en segundo plano
- ▶ No es necesario gestionar los clientes con contenedores
Deben conectar a `tcp://ipDelAnfitrión:8000`

9.2. Método automático: Ejemplo

- ▶ Crea subdir. broker, copia broker.js y este Dockerfile

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

- ▶ Crea subdir. worker, copia worker.js y este Dockerfile

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

- ▶ Crea subdir. client, copia client.js y este Dockerfile

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

9.2. Método automático: Ejemplo

- ▶ Dado el `docker-compose.yml` de la derecha, la orden:

`docker-compose up -d`

- ▶ Construye (*build*) cada imagen **solo si no existe** (*image* falla)
- ▶ Arranca una instancia de cada una en el orden correcto
- ▶ Puedes lanzar n instancias del servicio X con
`docker-compose up -d --scale X=n`
la orden -d hace que no se muestren en terminal
- ▶ Hay más órdenes en la guía

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

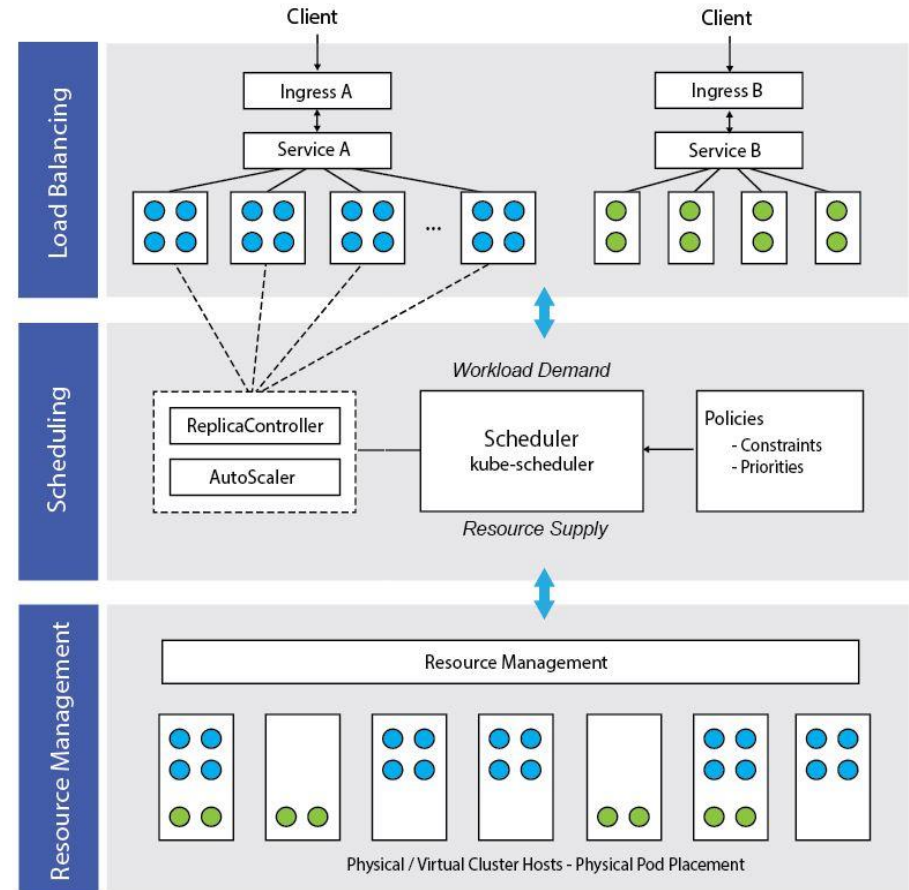


Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias

10. Múltiples componentes en distintos nodos

- ▶ Docker-compose se limita a componentes en un único nodo
- ▶ Pero queremos distribuir las instancias entre distintos nodos → la propuesta más conocida es **kubernetes**
- ▶ Es un orquestador de contenedores, pero no depende de Docker
 - ▶ La guía de este tema proporciona una descripción general de sus elementos



10. Múltiples componentes en distintos nodos

Kubernetes. Elementos principales:

- ▶ **Cluster** y nodo (físico o virtual)
- ▶ **Pod**: unidad más pequeña desplegable
 - ▶ incluye contenedores que comparten *namespace* y volúmenes
- ▶ **Controladores de replicación**: encargados del ciclo de vida de un grupo de pods,
 - ▶ asegurando que se encuentra en ejecución el número de instancias establecido,
 - escalando, replicando y recuperando pods
- ▶ **Controladores de despliegue**: actualizan la aplicación distribuida
- ▶ **Servicio**: define un conjunto de pods y la forma de acceso
- ▶ **Secretos** (gestión de credenciales)
- ▶ **Volúmenes** (persistencia)



Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias



II. Objetivos de aprendizaje

Al finalizar este tema, el alumno debe ser capaz de:

- ▶ Conocer con cierto nivel de detalle los aspectos a considerar en el despliegue de una aplicación distribuida
- ▶ Entender los problemas derivados de la existencia de dependencias, y algunas alternativas para tratarlos
- ▶ Entender el funcionamiento de una aproximación en el entorno de la nube, con conocimiento acerca de su operativa, posibilidades y limitaciones



Índice

1. Objetivos
2. Concepto de despliegue
3. Despliegue de un servicio
4. Automatización del despliegue
5. Despliegue en la nube
6. Contenedores
7. Docker
8. Creación de una imagen
9. Múltiples componentes en un nodo
10. Múltiples componentes en distintos nodos
11. Objetivos de aprendizaje
12. Referencias



12. Referencias

- ▶ Inversión de Control/Inyección de dependencias
 - ▶ <http://martinfowler.com/articles/injection.html>
 - ▶ <http://www.springsource.org/>
- ▶ www.docker.com (Website oficial de Docker)
 - ▶ docs.docker.com/userguide/ (**Documentación oficial**)
 - ▶ docs.docker.com/compose/ (Compose)
- ▶ github.com/wsargent/docker-cheat-sheet (Resumen de Docker)
- ▶ <http://kubernetes.io>
- ▶ 12factor.net/ (La metodología “*The twelve-factor app*”)