Tema 4. Despliegue de Servicios. Docker

Tecnologías de los Sistemas de Información en la Red

Índice

1. 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

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.

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
- Un trabajador
  - Debe conocer la ubicación del *broker* (IP y puerto *backend*)
  - Debe poseer una identidad única
- 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

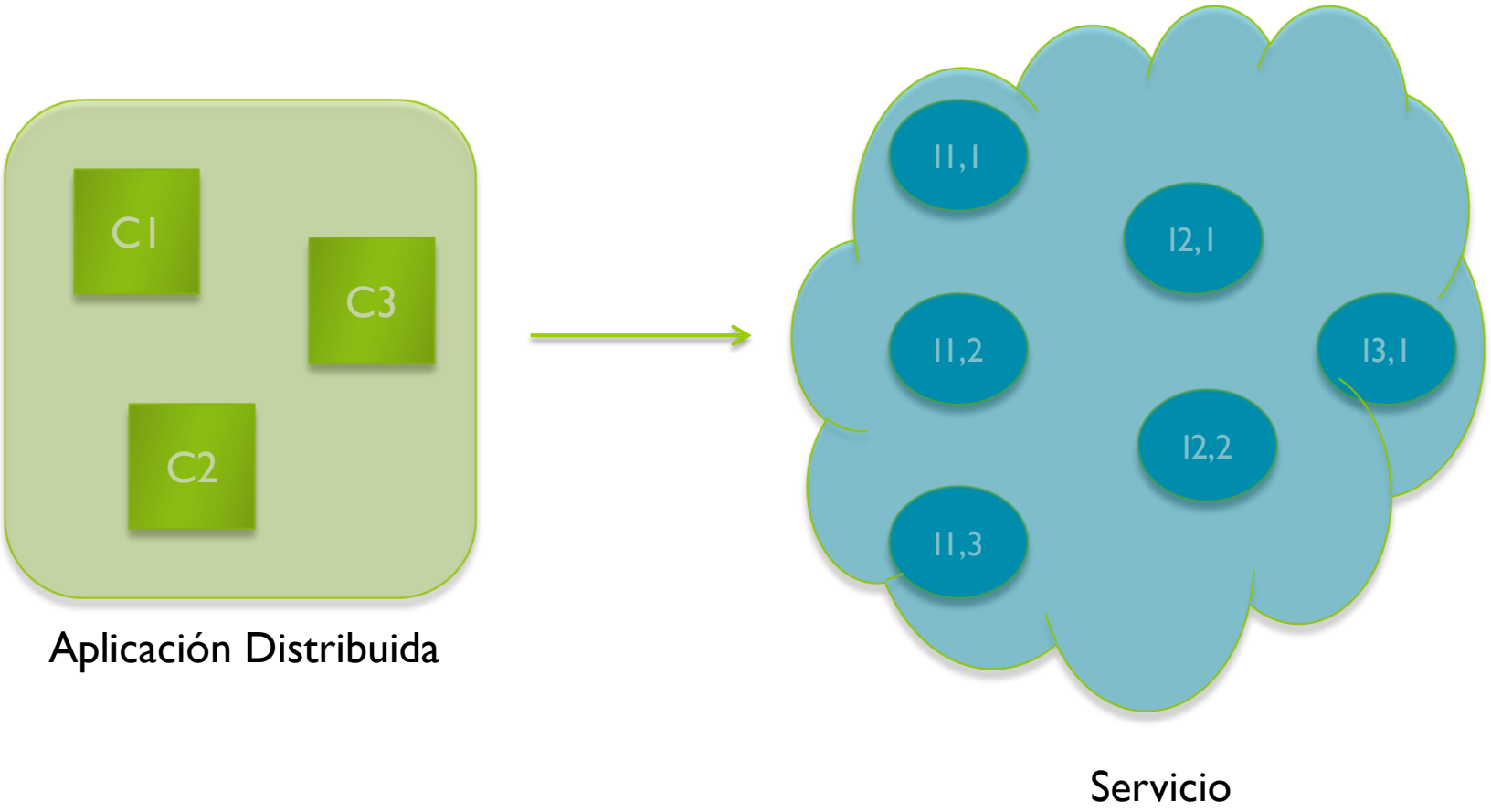
3. Despliegue de un servicio

- 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



Aplicación Distribuida

Servicio

Índice

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)

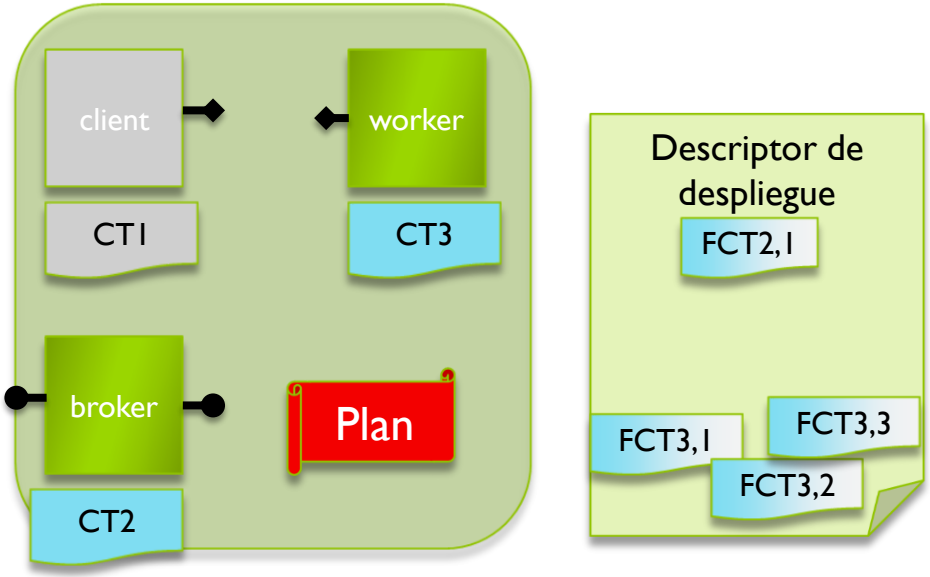
4. Automatización: Ejemplo

*[Aunque sea innecesario, aprovechamos el conocimiento del código manejado en el laboratorio 2, salvo que el **id** se calculará internamente]*

- Varias instancias de *client*
  - Argumento frontend (IP, port)
  - Dependencia respecto a *broker*
- Una instancia de *broker*
  - Argumentos frontend (IP_1, $port_1$) y backend (IP_2, $port_2$)
- Varias instancias de *worker*
  - Argumento backend (IP, port)
  - 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



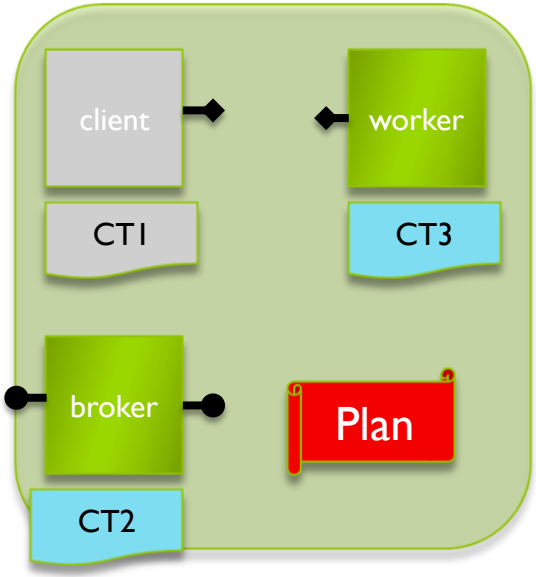
- Dependencia
- Endpoint
- Dependencia resuelta
- Endpoint resuelto
- Endpoint expuesto del servicio

CTi — Plantilla de configuración del componente i

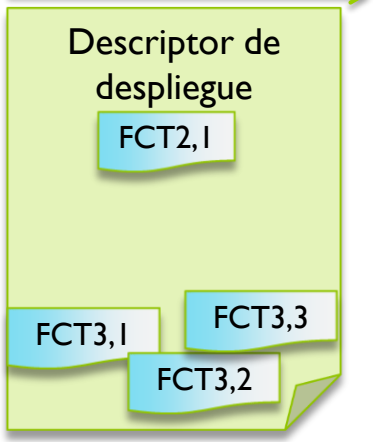
FCTi,j — Plantilla de configuración cumplimentada para la instancia j del componente i

4. Automatización: Ejemplo

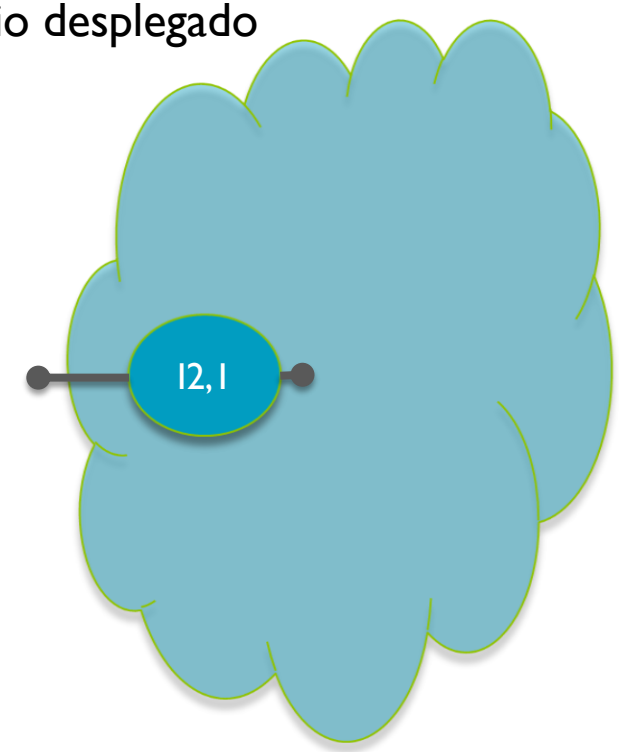
Aplicación distribuida



Despliegue



Servicio desplegado



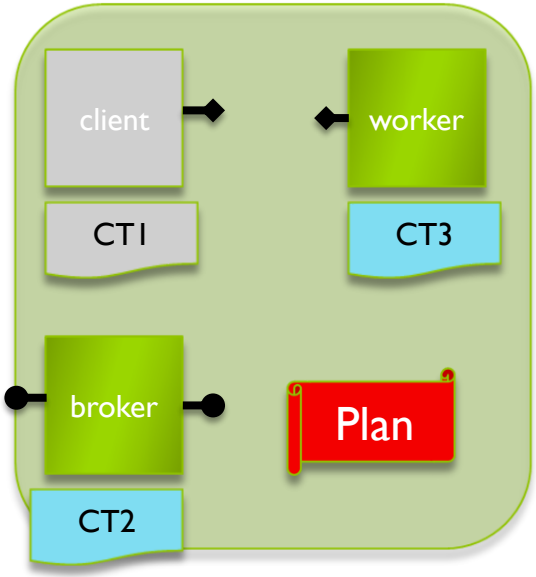
- Dependencia
- Endpoint
- Dependencia resuelta
- Endpoint resuelto
- Endpoint expuesto del servicio

CTi — Plantilla de configuración del componente i

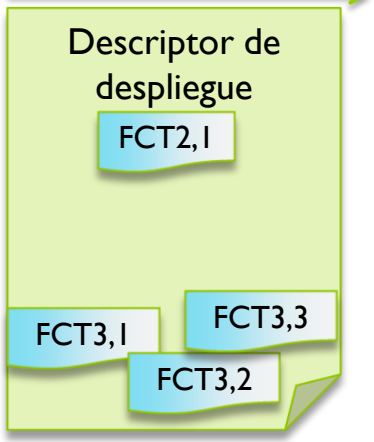
FCTi,j — Plantilla de configuración cumplimentada para la instancia j del componente i

4. Automatización: Ejemplo

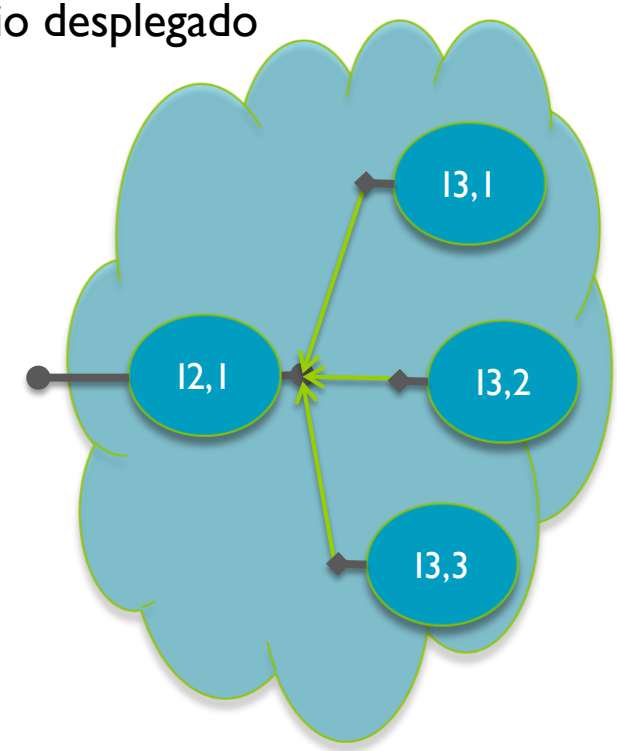
Aplicación distribuida



Despliegue



Servicio desplegado

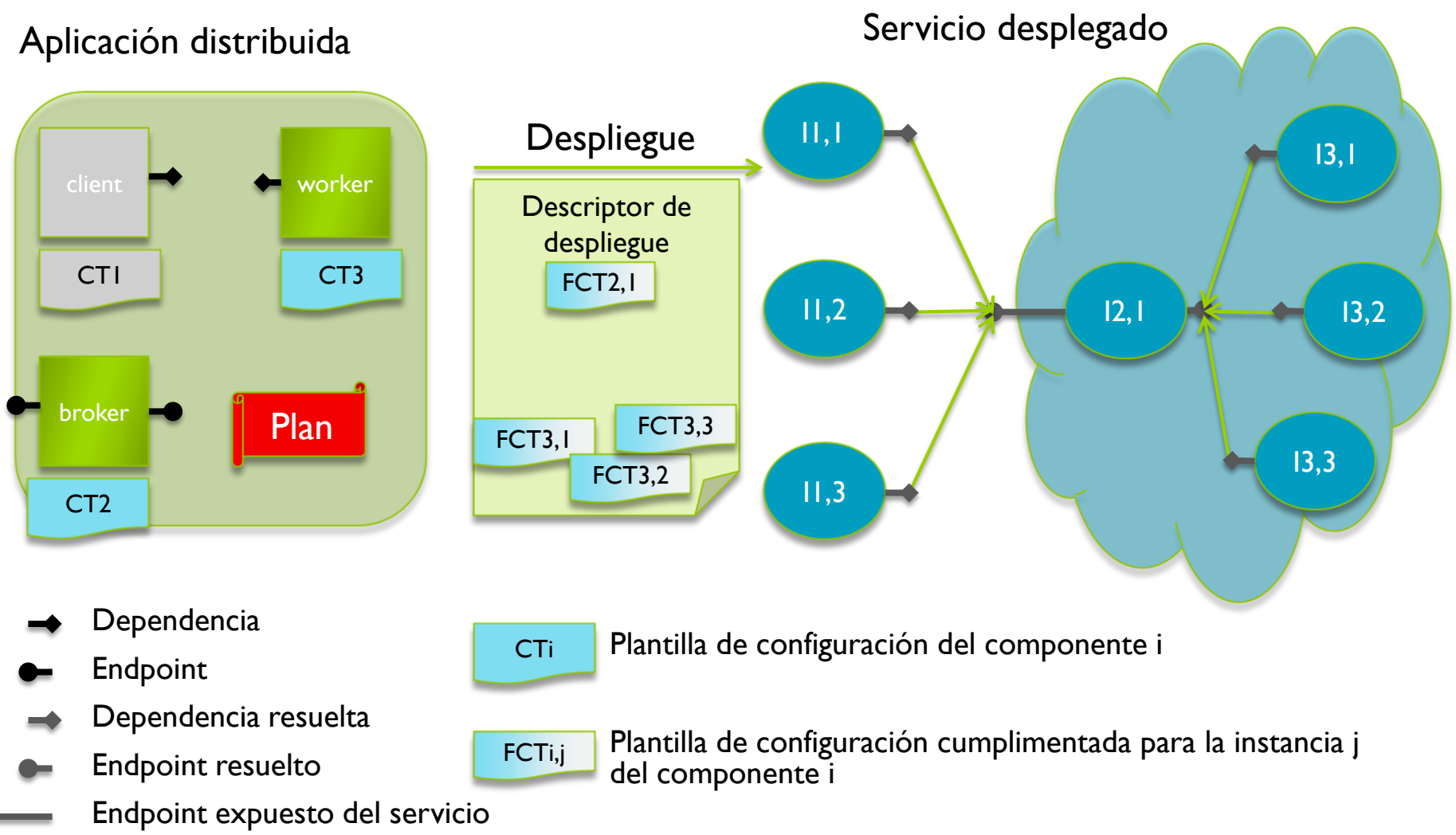


- Dependencia
- Endpoint
- Dependencia resuelta
- Endpoint resuelto
- Endpoint expuesto del servicio

CTi — Plantilla de configuración del componente i

FCTi,j — Plantilla de configuración cumplimentada para la instancia j del componente i

4. Automatización: Ejemplo



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

5.1. Despliegue en la nube: IaaS

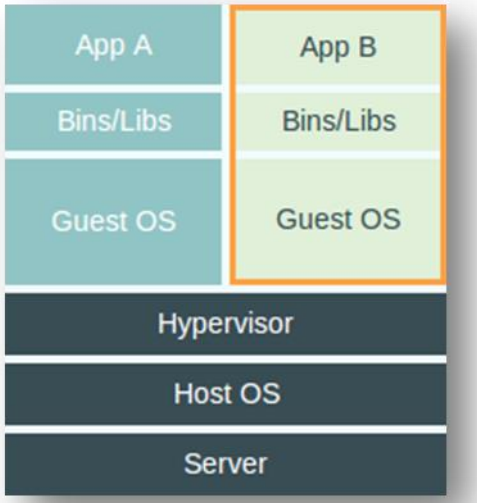
- Se basa en virtualización
  - Máquinas virtuales de distintos tamaños
  - Flexibilidad en la asignación de recursos
- Presenta limitaciones en el despliegue
  - Decisiones de asignación no automáticas (bajo nivel)
    - Número de instancias por componente, ubicación, tipo de MV
  - No permite elegir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

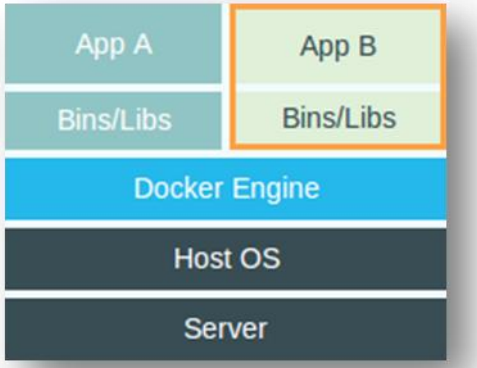
5.2. Despliegue en la nube: PaaS

- SLA como elemento central → parámetros del SLA para todos los componentes
  - Se persigue la automatización del despliegue
    - Planes de despliegue a partir del SLA
    - 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

- **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





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 900MB (SO) + 100MB (resto)
    - con MV: 100*(900MB + 100MB) = 100GB
    - con contenedores: 900MB + 100*100MB = 10.9GB
  - 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

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 `tsr-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 `tsr-zmq`
3. **Depósito:** lugar donde podemos dejar/obtener imágenes (espacio para compartir imágenes)
   - Ej.- Podemos subir la imagen `tsr-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 → `tsr-zmq`
  - `tsr-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 progInicial
```

- 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

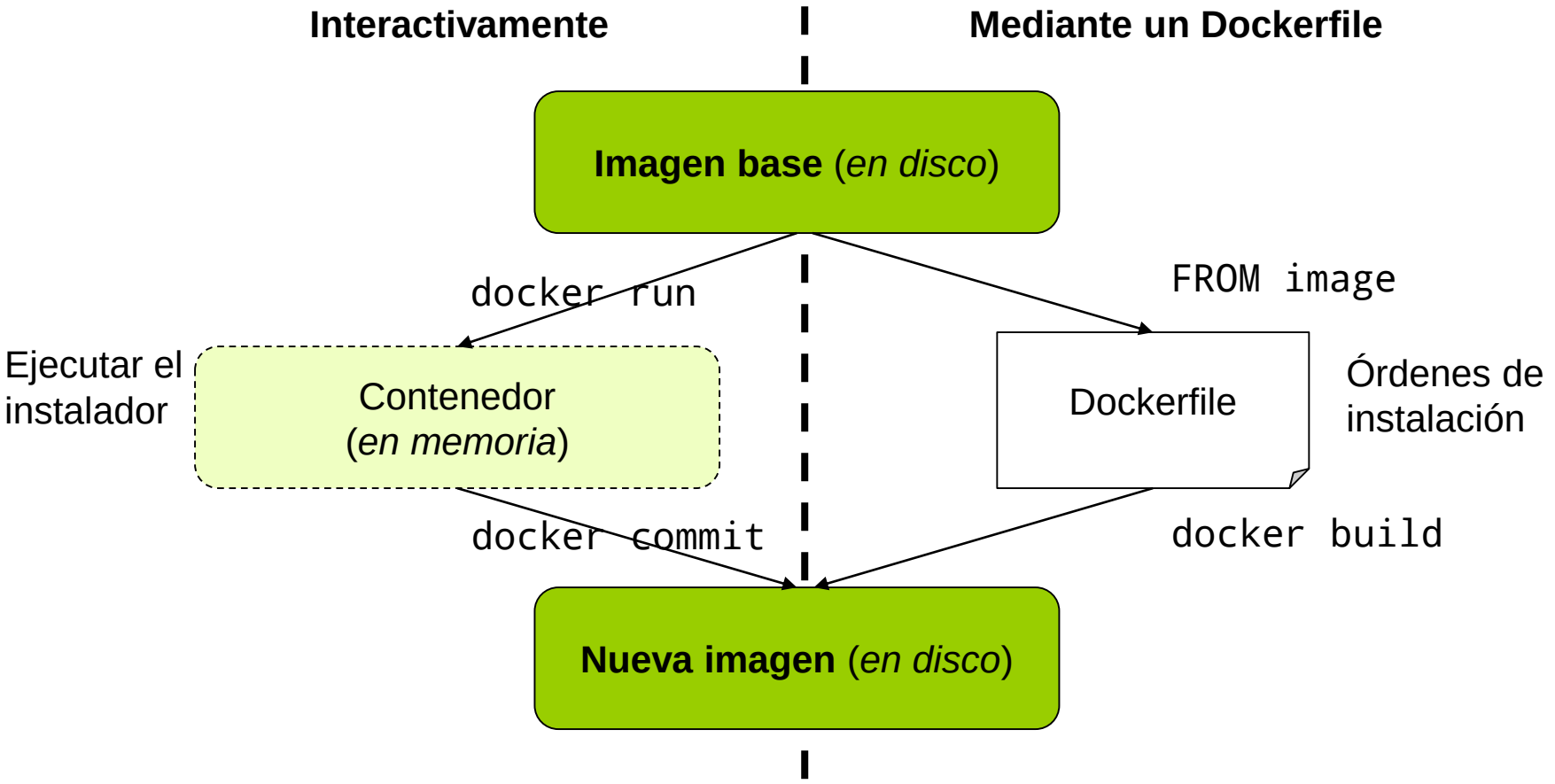
Grupo	Descripción
config	Gestión de configuraciones
container	Operaciones sobre contenedores
context	Contextos para el despliegue distribuido (k8s, …)
image	Gestión de imágenes
network	Gestión de redes
service	Gestión de servicios (contenedores instanciados desde la misma imagen) como parte de una aplicación 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

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:22.04 bash
```

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

```
$ cd /root
$ apt-get update -y
$ apt-get install curl ufw gcc g++ make gnupg -y
$ curl -sL https://deb.nodesource.com/setup_20.x | bash -
$ apt-get update -y
$ apt-get install nodejs –y; apt-get upgrade -y
$ npm init –y; npm install zeromq@5
$ exit
```

  - 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 Dockerfile:

```
FROM ubuntu:22.04
WORKDIR /root
RUN apt-get update -y
RUN apt-get install curl ufw gcc g++ make gnupg -y
RUN curl -sL https://deb.nodesource.com/setup_20.x | bash -
RUN apt-get update -y
RUN apt-get install nodejs -y
RUN apt-get upgrade -y
RUN npm init -y
RUN npm install zeromq@5
```

2. Nos situamos en el directorio de ese Dockerfile
3. Ejecutamos esta orden:
   - `docker build -t tsr-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

9. Múltiples componentes en un nodo. Ejemplo

- Ejemplo: Servicio implantado mediante un programa "broker", tolerante a fallos, y otro programa "worker", que podrá replicarse.
  - Idéntico a la versión tolerante a fallos de la práctica 2
  - Código del bróker (PRIMERA PARTE)

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
const ans_interval = 2000 // deadline to detect worker failure
lineaOrdenes("frontendPort backendPort")

let failed   = {}  // Map(worker:bool) failed workers has an entry
let working  = {}  // Map(worker:timeout) timeouts for workers executing tasks
let ready    = []  // List(worker) ready workers (for load-balance)
let pending  = []  // List([client,message]) requests waiting for workers
let frontend = zmq.socket('router')
let backend  = zmq.socket('router')

function dispatch(client, message) {
  traza('dispatch','client message',[client,message])
  if (ready.length) new_task(ready.shift(), client, message)
  else pending.push([client,message])
}
function new_task(worker, client, message) {
  traza('new_task','client message',[client,message])
  working[worker] = setTimeout(()=>{failure(worker,client,message)}, ans_interval)
  backend.send([worker,'', client,'', message])
}
```

9. Múltiples componentes en un nodo. Ejemplo

```
function failure(worker, client, message) {
  traza('failure','client message',[client,message])
  failed[worker] = true
  dispatch(client, message)
}
function frontend_message(client, sep, message) {
  traza('frontend_message','client sep message',[client,sep,message])
  dispatch(client, message)
}
function backend_message(worker, sep1, client, sep2, message) {
  traza('backend_message','worker sep1 client sep2
message',[worker,sep1,client,sep2,message])
  if (failed[worker]) return  // ignore messages from failed nodes
  if (worker in working) { // task response in-time
    clearTimeout(working[worker]) // cancel timeout
    delete(working[worker])
  }
  if (pending.length) new_task(worker, ...pending.shift())
  else ready.push(worker)
  if (client) frontend.send([client,'',message])
}

frontend.on('message', frontend_message)
 backend.on('message', backend_message)
frontend.on('error'   , (msg) => {error(`${msg}`)})
 backend.on('error'   , (msg) => {error(`${msg}`)})
 process.on('SIGINT' , adios([frontend, backend],"abortado con CTRL-C"))

creaPuntoConexion(frontend, frontendPort)
creaPuntoConexion( backend,  backendPort)
```

9. Múltiples componentes en un nodo. Ejemplo

- Código del worker y del cliente (basado en la práctica 2).
  - `workerReq` (izquierda) y `cliente`, con `id` autocalculado (no es un argumento)

```
const {zmq, lineaOrdenes, traza, error,
adios, conecta} = require('../tsr')
lineaOrdenes("brokerHost brokerPort")
let req = zmq.socket('req')
let id  = "W_"+require('os').hostname()
req.identity = id

conecta(req, brokerHost, brokerPort)
req.send(['','',''])

function procesaPeticion(cliente,
separador, mensaje) {
        traza('procesaPeticion','cliente
separador mensaje',[cliente, separador,
mensaje])

setTimeout(()=>{req.send([cliente,'',`${me
nsaje} ${id}`])}, 1000)
}
req.on('message', procesaPeticion)
req.on('error', (msg) =>
{error(`${msg}`)})
process.on('SIGINT', adios([req],"abortado
con CTRL-C"))
```

```
const {zmq, lineaOrdenes, traza, error, adios,
conecta} = require('../tsr')
lineaOrdenes("brokerHost brokerPort")
let req = zmq.socket('req')
let id  = "C_"+require('os').hostname()
req.identity = id

conecta(req, brokerHost, brokerPort)
req.send("C_"+require('os').hostname())

function procesaRespuesta(msg) {
        traza('procesaRespuesta','msg',[msg])
        adios([req], `Recibido: ${msg}.
Adios`)()
}
req.on('message', procesaRespuesta)
req.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([req],"abortado con
CTRL-C"))
```

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
  - 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`, `tsr.js` y este Dockerfile

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir broker
WORKDIR broker
COPY ./broker.js mybroker.js
EXPOSE 9998 9999
CMD node mybroker 9998 9999
```

  - 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 `workerReq.js`, `tsr.js` y el siguiente Dockerfile:

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir worker
WORKDIR worker
COPY ./workerReq.js myworker.js
# We assume that each worker is linked to the broker
# container.
CMD node myworker a.b.c.d 9999
```

  - **¡La dirección `a.b.c.d` debe averiguarse <u>después</u> de iniciar el broker!**
    - Podemos conocerla con `docker inspect` mientras se ejecuta el broker
- 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 `cliente.js`, `tsr.js` y el siguiente Dockerfile:

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir client
WORKDIR client
COPY ./cliente.js myclient.js
# We assume that each client is linked to the broker
# container.
CMD node myclient 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 portAnfitrion: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://ipDelAnfitrion:8000`

9.2. Método automático: Ejemplo

Aprovechamos la existencia de una configuración **`docker-compose.yml`** *por encima* de los Dockerfile, y que será capaz de sustituir las variables `$BROKER_HOST` y `$BROKER_PORT` por los valores adecuados en el momento de despliegue.

- Crea el subdirectorio `broker`, copia `broker.js`, `tsr.js` y este `Dockerfile`

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir broker
WORKDIR broker
COPY ./broker.js mybroker.js
EXPOSE 9998 9999
CMD node mybroker 9998 9999
```

  - NOTA: aunque se conozcan de antemano los puertos implicados, es buena práctica preparar la configuración de la manera más genérica posible. Observarás que dichos puertos se representan con variables en el worker y el cliente.

9.2. Método automático: Ejemplo

- Crea el subdirectorio `worker`, copia `workerReq.js`, `tsr.js` y este `Dockerfile`

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir worker
WORKDIR worker
COPY ./workerReq.js myworker.js
# We assume that each worker is linked to the broker
# container.
CMD node myworker $BROKER_HOST $BROKER_PORT
```

9.2. Método automático: Ejemplo

- Crea el subdirectorio `client`, copia `cliente.js`, `tsr.js` y este `Dockerfile`

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir client
WORKDIR client
COPY ./cliente.js myclient.js
# We assume that each client is linked to the broker
# container.
CMD node myclient $BROKER_HOST $BROKER_PORT
```

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
```

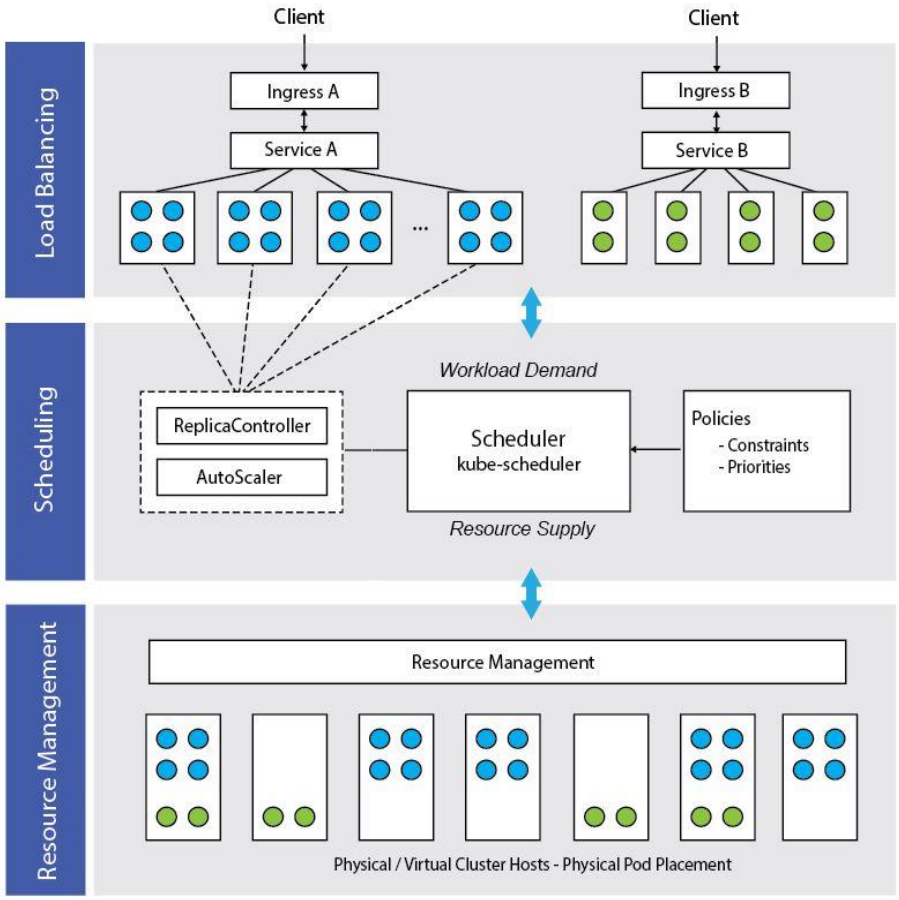
- Hay más órdenes en la guía

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

Índice

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

11. 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

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*")