TSR - Tema 4: **Despliegue de servicios**

Curso 2018/19

Contents

1. Objetivos

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

2. Concepto de despliegue

Despliegue = actividades que hacen que un sistema soft está preparado para su uso

→ actividades relacionadas con la instalación, activación, actualización y eliminación de componentes o del sistema completo

El despliegue de una aplicación distribuida es complejo:

- 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 (ej nuevas versiones, etc)

- Los componentes de la aplicación deben cooperar → existen dependencias entre ellos
- Los nodos pueden ser heterogéneos (ej distinto hard, SO, etc.), pero cada componente tiene ciertos requisitos para su ejecución
- Podemos tener requisitos adicionales de seguridad (privacidad, autenticación, etc.)

2.1. Ejemplo de despliegue

Utilizamos como ejemplo 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
- Podemos tener un número variable de instancias de cada componente (ej varios clientes, o varios workers). 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)

Dependencias y requisitos

- Un worker debe conocer la ubicación del broker, etc.
- Nuestros componentes requieren tener instalado Node y zmq

Para probar el patrón broker, realizamos manualmente varios pasos:

- 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 (javascript, nodo, zmq) con las versiones correctas
- Lanzar las instancias de los distintos componentes en el orden correcto (broker, workers, clientes)
  - En la línea de órdenes usada para lanzar cada instancia, indicar los argumentos necesarios (ej host y port para conectar con el broker, identidad, etc.)

3. Despliegue de un servicio

Desarrollamos sistemas distribuidos para ofrecer **servicios** (funcionalidad) a clientes remotos
Todo servicio establece un SLA (service level agreement)

- Definición funcional (qué hace)
- Rendimiento (qué capacidad tiene, tiempos esperados de respuesta, ..)
- Disponibilidad (en qué momentos puede accederse al servicio)
  - 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, y actualización del servicio
Define mecanismos para

- Instalación y activación.- ejecución del software
  - Resolver las dependencias del software (ej se necesitan determinadas bibliotecas, etc)
  - Configurar el software (instalarlo adecuadamente, versiones compatibles, parametrización, etc)
  - Determinar el número de instancias de cada tipo (ej. número de workers en map/reduce) y su reparto por los distintos nodos
  - Resolver las dependencias entre agentes (ej.- si uno escucha en un port, y otro se conecta, deben acordar el número de port)
  - Decidir el orden en que arrancan los componentes (ej broker, workers, clientes)
- Desactivación.- pasos para parar el sistema de forma ordenada
- Actualización.- pasos para reemplazar uno o más componentes por una nueva versión
- Adaptación
  - Fallo/recuperación de un agente (pasos a seguir para detección/recuperación)
  - Cambios en la configuración de los agentes (ej migración) sin detener el servicio
  - Escalado (reaccion ante cambios en la carga)

4. Automatización del despliegue

El despliegue puede ser muy complejo → un despliegue a gran escala no puede hacerse a mano
Necesitamos una herramienta que lleva a cabo el despliegue a partir de datos de entrada

- Configuración para cada componente (lista de parámetros de configuración + descripciones de dependencias)
  - A partir de este fichero, la herramienta genera una configuración específica para cada instancia de dicho componente
- Plan de configuración global
  - Plan de conexión entre componentes (lista endpoints expuestos, lista dependencias)
  - Decide donde colocar cada instancia
  - Enlace 'binding' de dependencias (asocia entre endpoints, incluyendo dependencias con servicios externos)

4.1. Ejemplo: patrón broker

- 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`

- El orden de arranque es broker, workers, clients
- Los endpoints son el frontend (externo) y el backend (interno)
- No hay dependencias respecto a servicios externos

Resolución de dependencias. Opciones:

1. El código define la forma de resolver las dependencias (ej.- leyendo datos de un fichero, o bien recibiendo datos en socket)
    - bajo nivel
2. Inyección de dependencias. El código de la aplicación expone nombres locales para sus interfaces relevantes, y 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

5. Despliegue en la nube

5.1. IaaS (Infraestructure as a Service)

Se basa en virtualización (flexibilidad en la asignación de recursos) → Máquinas virtuales de distintos tamaños

Limitaciones en el despliegue

- Decisiones de asignación (num instancias de cada componente, ubicación, tipo de MV) de bajo nivel, no automatizadas
- Sin posibilidad de elegir características de red (latencia, throughput)
- Modelo de fallo insuficiente (modos de fallo no son realmente independientes, ayuda limitada a la recuperación)

5.2. PaaS (Platform as a Service)

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
    - Automatización limitada (despliegue inicial, pero no gestión del SLA ni actualizaciones)
    - Microsoft Azure es uno de los mas evolucionados

6. Contenedores

Aprovisionamiento = reservar la infraestructura necesarias para una aplic 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 servidor anfitrión

    Suponemos el uso de contenedores en lugar de MV

- Menor flexibilidad
  - El software de la instancia ha de ser compatible con el SO del anfitrion
  - Debe configurarse (ej soft adicional) e inicializarse correctamente
  - El aislamiento entre contenedores no es perfecto
- Utiliza muchos menos recursos
  - Ej suponemos que la imagen de una instancia requiere 1GB, de los cuales 900MB son para el SO
    * con MV, para 100 instancias 100*1GB = 100GB
    * con contenedores, 0.9MB + 100*0.1MB = 10.9GB
  - ahorramos espacio y tiempo (ej para instalar dicha imagen)
- Mayor facilidad de despliegue (fichero de configuración)
- Aplicable en casi todos los escenarios

Herramienta.- sistema de contenedores **Docker**

- Es el mas conocido
  - 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 sólo lectura para compartición entre contenedores
  - Permite cooperación en el desarrollo mediante depósitos públicos
  - Asumimos que el SO del huesped es Linux (aunque existe Docker para Windows)
- Reservamos el despliegue manual para pruebas simples

7. Docker

Componentes

1. Imagen.- plantilla de sólo lectura con las instrucciones para crear un contenedor
   - Una imagen se define a partir de otra mas básica, a la que añadimos determinadas instrucciones. Por ejemplo:
     - Tenemos imágenes predefinidas para las distintas distribuciones Linux (ej imagen `Centos:7.4.1708`)
     - Añadimos las instrucciones para instalar node → creamos imagen `centos-nodejs`
     - Añadimos las instrucciones para instalar zmq → creamos la imagen `centos-zmq`
     - A partir de `centos-zmq`, creamos imágenes para cada componentes (client,broker,worker)

2. Contenedor.- Conjunto de recursos que necesita una instancia para ejecutarse. Se crea al ejecutar una imagen

    - Ej al construir un contenedor a partir de la imagen centos-zmq se puede ejecutar sobre el mismo una instancia de client, broker o worker

3. Depósito.- lugar donde podemos dejar/obtener imágenes (espacio para compartir imágenes)

    - Ej.- Podemos subir la imagen centos-zmq, y cualquiera puede bajarla y usarla para crear contenedores

Para trabajar con esos componentes se usan órdenes desde consola

- La estructura general es `docker accion opciones argumentos` (opción x como -x)
  - Bajar imagen desde el depósito `docker pull imagen`
  - Crear e iniciar un contenedor desde una imagen `docker run opciones imagen programaInicial`
    - Ej `docker run -i -t centos /bin/bash` descarga la imagen `centos`, crea el contenedor, reserva sistema de ficheros, reserva interfaz de red y dirección IP interna, y ejecuta `/bin/bash`
      - Las opciones `-i -t` sirven para modo interactivo (la consola queda abierta y conectada al contenedor)
  - Modificar contenedor (mediante órdenes en modo interactivo desde consola)
  - Crear nueva imagen a partir del estado actual del contenedor `docker commit nombreContenedor nombr`
  - Subir imagen al depósito `docker push imagen`

Otas órdenes permiten:

- Arrancar (`start`), detener (`stop`), o reiniciar (`restart`) la ejecución del contenedor
- Eliminar un contenedor ya detenido (`rm`)
- Obtener información sobre las imágenes o los contenedores activos (`images`, `ps`)
- Obtener información sobre una imagen (`history`)
- Obtener información sobre un contenedor (`inspect`)
- Etc.

Un contenedor puede acceder a recursos del anfitrión

- Sistema de ficheros `docker run ... -v pathAnfitrion:pathContenedor`
- Puerto `docker run ... -p portAnfitrion:portContenedor`

Ejemplo: crear la imagen que permite ejecutar programas nodejs usando zmq

- `docker run -i -t centos:7.4.1708 bash` crea un contenedor a partir de imagen centos, lanza interprete de órdenes bash en modo interactivo (espera órdenes)
- Lanzamos las siguientes órdenes de forma interactiva

```
curl --silent --location  https://rpm.nodesource.com/setup_8.x | bash
yum install -y nodejs
yum install -y epel-release
yum install -y zeromq-devel make python gcc-c++
yum install -y zmq
exit
```

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

8. Creación de una imagen con Dockerfile

Hemos visto que podemos tomar una imagen base, crear el contenedor, modificarlo, y guardar ese estado como una nueva imagen

```
docker run ....     cambios en el contenedor ...  docker commit ...
```

Alternativa: crear una nueva imagen a partir de las instrucciones de un fichero de texto denominado Dockerfile

```
docker build -t nombreNuevaImagen pathDockerfile
```

Ej.- para construir una imagen basada en centos con soporte para nodejs y zmq, escribimos el texto en un fichero llamado Dockerfile

```
FROM centos:7.4.1708
RUN curl --silent --location  https://rpm.nodesource.com/setup_8.x | bash
RUN yum install -y nodejs
RUN yum install -y epel-release
RUN yum install -y zeromq-devel make python gcc-c++
RUN yum install -y zmq
```

nos situamos en en directorio donde está Dockerfile y ejecutamos `docker build -t tsr1718/centos-zmq .`
Respecto al Dockerfile

- Cada línea empieza con una instrucción, que por convención se escribe en mayúsculas
- La primera instrucción (primera línea) debe ser FROM (indica qué imvagen se toma como base para construir la nueva)
- La instrucción RUN ejecutar una orden para el shell
- Son posibles otras instrucciones
  - ADD origen destino
    copia ficheros de un lugar a otro. origen = URL, directorio, o archivo, destino = ruta en el contenedor Si el origen es un directorio, lo copia completo. Si es un fichero comprimido, lo expande al copiar
  - COPY origen destino igual que ADD, pero no expande los ficheros comprimidos
  - CMD orden arg1 arg2 proporciona valores por defecto para la ejecución del contenedor
  - EXPOSE puerto indica el puerto en el que el contenedor atenderá peticiones
  - ENTRYPOINT orden arg1 arg2 Al crear el contenedor, ejecuta esta orden (termina al finalizar la orden)
  - 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 contenedorSólo debería haber como máximo una orden `CMD` o `ENTRYPOINT` (si hay mas, sólo ejecuta el último)

9. Múltiples componentes en el mismo nodo

Cuando hay varios componentes tenemos dependencias entre imágenes

- La imagen del cliente necesita conocer la URL del frontend (IP y port)
- La imagen del worker necesita conocer la URL del backend (IP y port)

No tenemos esa información hasta que arranca el broker

1. Una vez lanzado el broker en su contenedor, obtenemos la IP del contenedor
2. Modificamos manualmente esos valores en los Dockerfile de clientes y Workers para crear correctamente sus imágenes
3. Lanzamos workers y luego clientes

Este esquema 'manual' no escala bien para desarrollos grandes. Necesitamos:

- Un lenguaje para describir componentes, propiedades y relaciones → plan de trabajo
- Una herramienta que ejecute el despliegue a partir de ese plan de trabajo

9.1. Ejemplo

Suponemos que previamente hemos creado una imagen `tsr1718/centos-zmq` (con soporte para nodeJS y zmq)

- Sobre esa imagen se pueden ejecutar programas nodeJS que usen 0MQ

9.1.1. Método manual

- broker
  - En un directorio copiamos el código broker.js y el siguiente fichero llamado Dockerfile

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./broker.js /zmq/broker.js
WORKDIR /zmq
EXPOSE 8059 8060
CMD node broker
```

    - El frontend es el port 8059 y el backend el 8060
  - En este directorio ejecutamos: `docker build -t broker .`
  - Para crear la imagen lanzamos el broker con `docker run -d broker`
    - la opción -d lanza el contenedor en segundo plano
  - averiguamos la URL del contenedor que ejecuta el broker (suponemos que es a.b.c.d)
    - `docker ps -a` para conocer el ID del contenedor
    - `docker inspect ID` para obtener su dirección IP
- worker
  - En otro directorio copiamos el código `worker.js` y el siguiente fichero llamado Dockerfile

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
```

```
COPY ./worker.js /zmq/worker.js
WORKDIR /zmq
CMD node worker tcp://a.b.c.d:8060
```

  - En este directorio ejecutamos `docker build -t worker .` para crear la imagen
  - Lanzamos el worker con `docker run -d worker` (podemos lanzar tantos como consideremos necesario)

- client (si los clientes son locales)

  - En otro directorio copiamos el código `client.js` y el siguiente fichero llamado Dockerfile

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./client.js /zmq/client.js
WORKDIR /zmq
CMD node worker tcp://a.b.c.d:8059
```

  - En este diectorio ejecutamos `docker build -t client .` para crear la imagen
  - Lanzamos el cliente con docker run -d client (podemos lanzar tantos como consideremos necesario)

- client (si los clientes son remotos)

  - Lanzamos el broker con `docker run -p 8000:8059 -d broker`
    - La opción -p portAnfitrion:portWorker permite acceso al frontend desde el exterior (port 8000 del anfitrion)
    - La opción -d lanza el contenedor en segundo plano
  - No es necesario gestionar los clientes con contenedores
  - Deben conectar a `tcp://ipDelAnfitrion:8000`

9.1.2. De forma automática

Usando docker-compose. Está limitado a contenedores en el mismo equipo anfitrión Creamos un directorio con 3 subdirectorios (broker, worker, client). Nos situamos sobre dicho directorio

- Copiamos en el subdirectorio broker el código `broker.js` y el siguiente Dockerfile

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./broker.js /zmq/broker.js
WORKDIR /zmq
EXPOSE 8059 8060
CMD node broker
```

- Copiamos en el subdirectorio worker el código `worker.js` y el siguiente Dockerfile

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./worker.js /zmq/worker.js
WORKDIR /zmq
CMD node worker $BROKER_URL
```

- Copiamos en el subdirectorio client el código `client.js` y el siguiente Dockerfile

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./client.js /zmq/client.js
WORKDIR /zmq
CMD node client $BROKER_URL
```

Copiamos en este directorio el siguiente fichero, denominado docker-compose.yml

```
version: '2'
services:
 cli:
     image: client
     build: ./client/
     links:
         - bro
     environment:
         - BROKER_URL=tcp://bro:8059
 bro:
     image: broker
     build: ./broker/
     expose:
         - "8059"
         - "8060"
 wor:
     image: worker
     build: ./worker/
     links:
         - bro
     environment:
         - BROKER_URL=tcp://bro:8060
```

Ejecutamos `docker-compose up -d` → construye las 3 imágenes, y arranca una instancia de cada una en el orden correcto

Con una orden similar se pueden lanzar varias instancias: `docker-compose up -d --scale SERVICE=NUM`

10. Múltiples componentes sobre distintos nodos

- Docker-compose se limita a componentes en un único nodo
- Pero queremos distribuir las instancias entre distintos nodos → la propuesta mas conocida es **kubernetes**
  - Es un orquestador de contenedores, pero no depende de Docker

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

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)
  - docs.docker.com/engine/swarm/ (Modo Swarm)
- github.com/wsargent/docker-cheat-sheet (Resumen de Docker)
- http://kubernetes.io
- 12factor.net/ (La metodología 'The twelve-factor app')