TSR - PRÁCTICA 3
CURSO 2014/15

DESPLIEGUE

El laboratorio 3 se desarrollará a lo largo de tres sesiones. Sus objetivos principales son:

> "Que el estudiante comprenda algunos de los retos que conlleva el despliegue de un servicio multi-componente, presentándoles un ejemplo de herramientas y aproximaciones que puede emplear para abordar tales retos"

Para alcanzar este objetivo, proponemos la siguiente secuencia:

1. Construir las imágenes de los componentes que se desplegarán mediante *Docker*.
2. Analizar un programa de despliegue para servicios cuyos componentes son programas **`nodejs`**, empaquetados como imágenes **`Docker`**.
3. Partiendo del anterior punto 2, implementar un servicio de despliegue para vuestra aplicación resultante del laboratorio 2.
4. Desde el punto 3, implementar un servicio de generación de imágenes sencillas.
5. Modificar el servicio generador de imágenes, añadiendo un componente caché para evitar la repetición de operaciones ya realizadas.
6. Modificar el servicio generador de imágenes para aumentar la escalabilidad potencial del servicio.

En este documento, la **parte 1** se dedica al manejo básico de Docker, herramienta que usaremos como base de nuestra estrategia de despliegue.

La **parte 2** se centra en la descripción de los servicios y despliegues, así como en el propio despliegue de los servicios definidos en función de las líneas maestras que se presentan aquí.

Por último, en la **parte 3** se presentan las especificaciones para modificar y desplegar el servicio de generación de imágenes.

La parte 1 debería cubrirse en la primera sesión de laboratorio, la parte 2 debe realizarse en la segunda, y el tiempo restante se debe dedicar a la parte 3.

Como es habitual en TSR, debe emplearse Git como ayuda al desarrollo de las actividades de laboratorio, así como para la entrega de los resultados de cada equipo de laboratorio.

Los resultados de esta práctica se organizarán de la siguiente forma en el depósito:

- El directorio principal **`Lab3`** que contiene el depósito se divide en tres subdirectorios

- Directorio **`Components`**
  - Contiene un directorio por cada **componente** a desplegar como parte de cualquiera de los servicios. El directorio contiene el código del componente, como un paquete estándar nodejs. Además, se incluye un **`Dockerfile`** que especifica cómo empaquetarlo como imagen docker.
- Directorio **`Services`**
  - Contiene un subdirectorio por **servicio**. Cada directorio contiene un archivo de configuración **`index.js`**, que contiene el objeto de configuración para el servicio.
- Directorio **`Deployments`**
  - Contiene un directorio por cada despliegue que se haya especificado. Dentro de estos directorios hay un archivo index.js con la configuración del despliegue.
- Directorio **`Deployers`**
  - Contiene un directorio por cada **`desplegador`** que se haya implantado. Cada directorio desplegador contiene el código de un **programa desplegador**. Este programa consume uno de los directorios de despliegue que describe el despliegue de un servicio de entre los del directorio **`Services`**, y crea todas las instancias de los componentes que necesite el servicio, asegurando que se configuran adecuadamente entre ellas.

Los materiales necesarios para este laboratorio son:

1. El resultado de vuestro Lab2
2. Los contenidos del depósito **`git@gitlab.dsic.upv.es:tsr/lab3support.git`**, al que podéis acceder de la forma habitual.

CONTENIDOS

1 INTRODUCCIÓN

Los servicios son el resultado de la ejecución de una o varias instancias de cada uno de los componentes software empleados para implementarlo.

Uno de los problemas en el momento del despliegue de un servicio es el empaquetamiento de cada uno de sus componentes de manera que la instanciación de esos componentes sea repetible de forma determinista, y que la ejecución de las instancias de componentes se aísle de la ejecución del resto de instancias de cualquier componente.

Otro problema a abordar consiste en la configuración de cada una de las instancias a desplegar. Esto no sólo afecta a los parámetros de los algoritmos internos implementados por el componente, sino que también afecta a la manera de configurar las dependencias respecto a otros componentes, que, normalmente, conduce a algun tipo de establecimiento de canales de comunicación entre diferentes instancias.

En este laboratorio exploramos formas de construir desplegadores capaces de ofrecer una solución sencilla para ambos problemas.

Nos enfrentamos al primer problema con la ayuda del framework **Docker**. Tal y como se ha estudiado en el seminario, **Docker** permite preparar toda la pila software necesaria para la instanciación de un componentes, de manera adecuada y fiable, sin interferir con ninguna de las otras aplicaciones que se puedan encontrar instaladas en el sistema operativo del anfitrión.

Además, la pila para el componente se encuentra preparada para poder lanzarse dentro de un contenedor Linux, garantizando su transportabilidad de un sistema a otro, así como un buen grado de aislamiento de la instancia en ejecución, contribuyendo de manera apreciable a la repetibilidad de su despliegue.

Para resolver adecuadamente el segundo problema se necesitaría prepara un sistema de *inyección de dependencias*, ocultando así a los componentes cómo se seleccionan tales dependencias, así como los detalles de bajo nivel del protocolo usado, y dejando esta decisión completamente en manos del desplegador.

No obstante, para poder resolver este problema en el tiempo disponible para este laboratorio, adoptaremos una aproximación más tradicional, en la que la configuración se implementa como un objeto, almacenado en un modulo JavaScript de nodejs.

2 PARTE 1: PRIMEROS PASOS. USO DE DOCKER Y PREPARACIÓN DE LAS IMÁGENES PARA NUESTROS COMPONENTES DEL LABORATORIO 2

Todos nuestros componentes se van a ejecutar en Nodejs, y usarán ZeroMQ. Además, nuestra aplicación objetivo requerirá que dispongamos del software gráfico ImageMagick.

Incluso aunque no todos nuestros componentes necesitarán ImageMagick, en aras a la simplicidad prepararemos una imagen común, con todos los componentes anteriormente

mencionados, como parte de la pila de software en la que se basarán todos nuestros componentes.

Nuestra primera tarea será la preparación de la imagen base para el resto del laboratorio 3.

Hemos preparado un depósito con el software que se necesitará a lo largo de la práctica. Puedes obtenerlo mediante la orden:

```
> git clone git@gitlab.dsic.upv.es:tsr/lab3support.git
```

Tras ejecutar esta orden, el directorio `lab3support` contendrá varias carpetas, así como una copia `pdf` de este enunciado. El archivo `README.md` explica cuál es el contenido de cada directorio. Uno de ellos se llama `tsrImage`.

Para producir el material de la primera parte del laboratorio, crea dentro del depósito `Lab3` un directorio llamado `baseimages`. La primera tarea es colocar una copia de `tsrImage` en este directorio.

Dentro de `tsrImage` se pueden observar un par de archivos. Uno es un `Dockerfile`. El otro es un script ejecutable `buildimage.sh`, empleado para indicarle a `docker` cómo construir la imagen base.

Un vistazo al `Dockerfile` dentro de `tsrImage`:

```
01:   FROM base/archlinux
02:
03:   MAINTAINER Jose Bernabeu <josep@dsic.upv.es>
04:
05:   RUN pacman -Syu --noconfirm --needed gcc make imagemagick zeromq python2
   nodejs \
06:     &&  rm /var/cache/pacman/pkg/* \
07:     &&  pacman -Sc  --noconfirm \
08:     &&  mkdir /app \
09:     &&  ln -s /usr/bin/python2 /usr/bin/python
10:
11:
12:   ONBUILD ADD package.json /tmp/
13:   ONBUILD RUN cd /tmp && npm install && rm package.json
14:   ONBUILD ADD . /app
15:   ONBUILD RUN cd /app && rm -rf node_modules && mv /tmp/node_modules .
16:   WORKDIR /app
17:
18:
19:
20:   ENTRYPOINT ["/usr/bin/npm", "start"]
```

Como se puede ver, el archivo Dockerfile indica a docker que construya una imagen a partir de una imagen base "archlinux". Después continúa actualizando el sistema de instalación de paquetes de archlinux (`pacman -Syu`), e, inmediatamente a continuación, instala los paquetes que va a necesitar:

- `zeromq`: las bibliotecas binarias para el middleware `0MQ`.
- `nodejs`: nuestro soporte de programación base

- python2: necesario para npm para empaquetar bibliotecas binarias. Necesario para que npm instale zmq.
- gcc y make: también necesarios para la operación npm install de los paquetes binarios (como zmq).

Inmediatamente a continuación, se limpian los paquetes intermedios para ahorrar espacio en la imagen resultante. Después se crea un directorio /app en el que planeamos colocar todo el código de la aplicación para cada instancia. Finalmente, creamos un enlace simbólico de python a python2, para que python2 se considere como versión por defecto del intérprete python, tal y como necesita npm.

Las siguientes instrucciones del Dockerfile están prefijadas por la directiva ONBUILD. Esta directiva ordena que docker ejecute la orden que sigue a la directiva en su misma línea cuando encuentre algún Dockerfile posterior que tome como base la imagen resultante de este Dockerfile.

Aunque debemos especificar las directivas ONBUILD en la construcción de nuestra imagen base, sus efectos no entran en acción hasta que construyamos las imágenes para nuestros componentes basados en nodejs, tal y como veremos más tarde. Mediante estas instrucciones, al crear la imagen de cada componente que se basa en la imagen base que construimos ahora, instalaremos el código para el componente que estamos construyendo, así como las dependencias para ese componente. Esto se consigue de la siguiente forma:

- ADD package.json /tmp/ copia el archivo package.json desde el directorio del anfitrión al /tmp del contenedor.
- RUN cd /tmp && npm install && rm package.json primero cambia al directorio /tmp, después ejecuta npm install, incorporando todas las dependencias declaradas en el archivo packages.json dentro del directorio node_modules en /tmp, y justo a continuación elimina el fichero package.json de /tmp.
- ADD . /app copia todos los ficheros del directorio del host al directorio /app del contenedor.
- Tras mover el directorio /app, el último ONBUILD elimina el directorio node_modules procedente del anfitrión, y en su lugar mueve el directorio node_modules procedente de /tmp, que contiene todas las dependencias necesarias. De esta manera evitamos copiar al contenedor binarios compilados para el anfitrión, que podrían no ser ejecutados correctamente.

Tras las instrucciones gobernadas por ONBUILD, ordenamos a docker que tome /app como directorio de trabajo. /app es la carpeta en la que reside el código para nuestros componentes. De esta forma se asegura que cuando se lance una instancia del contenedor basada en la imagen resultante, el proceso creado tendrá como directorio de trabajo /app.

La última instrucción es ENTRYPOINT ["/usr/bin/npm", "start"], que provoca que docker ejecute "/usr/bin/npm start" en cada ocasión en que deba instanciar esta imagen para un nuevo contenedor. Esta orden se ejecuta dentro del directorio /app que contiene todo el código de la aplicación del componente. El efecto de ejecutar tal orden es que, por defecto,

"node server.js" se ejecuta en el directorio /app del contenedor, lanzando nuestro componente.

Construyamos nuestra instancia base. Para ello pasamos al directorio tsrImage y ejecutamos las instrucciones siguientes:

```
> docker build –t tsir/baselab3 .
```

Puede conseguirse el mismo efecto ejecutando el script buildimage.sh que se encuentra en el mismo directorio.

Tras dejar que finalice la orden anterior, deberíamos poder ver nuestra nueva imagen ejecutando

```
> docker images
```

Una de las entradas listadas debe tener como nombre tsir/baselab3.

2.1 Componentes e imágenes de componentes

Nuestro próximo paso será preparar un conjunto de componentes y sus imágenes correspondientes, dejándolas preparadas para la segunda parte del laboratorio.

Continúa y copia el directorio Components de la carpeta lab3support a tu depósito Lab3. Encontramos dos subdirectorios dentro de Components: balancer y frontend. Usamos estos componentes como ejemplos para mostrar cómo especificar un componente.

Hablando en general, un componente es un conjunto autocontenido de ficheros, conteniendo código y material auxiliar, con dependencias de código bien definidas sobre un entrono base que ayude en su ejecución. Por tanto, para especificar un sistema de componentes, es necesario especificar también el entorno en el que se supone que se han de ejecutar los componentes de ese sistema, y cómo deberían construirse esos componentes para poder ejecutarse en dicho entorno.

Nuestro entorno de ejecución de componentes es un contenedor docker ejecutando archlinux, y enriquecido con nodejs, zeromq, …

Además, la construcción de un ***component*** es algo tan sencillo como colocar los ficheros de código (además de material adicional) en un directorio, de acuerdo con las siguientes reglas:

1. El código principal del componente se coloca en el archivo server.js en la raíz de este directorio.
2. El directorio debería contener también un subdirectorio de nombre config. Este subdirectorio debería contener un fichero default.js, cuyos contenidos se describirán en breve.
3. El directorio debe contener un fichero package.json que describa el componente. El archivo package.json debe incluir al menos:
    a. El nombre del componente
    b. Las dependencias del código nodejs del componente
        i. Una de las dependencias DEBE ser el paquete "config" de nodejs.

c. El módulo que se ejecutará cuando se necesite el paquete mediante la directiva `require` propia de `nodejs`. Además, en nuestro caso, el valor de este atributo siempre debe ser "`config/default.js`".

Con un vistazo a los directorios `balancer` y `frontend` observamos que ambos cumplen con la anterior definición. Profundicemos un poco en el aspecto de la descripción y configuración de componentes. A continuación se muestra el archivo `package.json` para el componente `balancer`:

```
01:   {
02:         "name": "balancer",
03:         "main": "config/default.js",
04:         "dependencies": {
05:             "zmq": "*",
06:             "config": "*"
07:         }
08:   }
```

Podemos observar, tal y como esperábamos, que su formato es JSON. En este caso las dependencias solamente incluyen los paquetes "`zmq`" y "`config`". Anunciamos que el módulo principal es "`config/default.js`", por lo que cualquier `require` en este paquete ejecutará `config/default.js`.

2.1.1 Configuración y descripción de componentes

Para desplegar adecuadamente un componente creando una o varias instancias, necesitamos saber cuáles son sus puntos de configuración, de manera que podamos rellenarlos en un despliegue.

Con dicho propósito empleamos el fichero `config/default.js`. Este archivo es un módulo `nodejs` que exporta un objeto de configuración sencillo. El objeto se ajusta a las siguientes reglas:

1. Contiene cuatro atributos: "`requires`", "`provides`", "`export`" y "`parameter`", cada uno de los cuales toma otro objeto como valor. Si ese objeto está vacío, el atributo puede omitirse de `default.js`.
    a. Para cada uno de los objetos, cada valor de sus atributos representa el valor por defecto de ese atributo, que puede (y suele) modificarse durante el despliegue.
2. El objeto `requires` contiene un atributo por cada URL a un socket `zmq` perteneciente a otra instancia de un componente al que éste necesite conectar.
3. El objeto `provides` contiene un atributo por cada socket `zmq` en el componente que se necesita asociar. El valor del atributo debe ser el número del puerto al que asociar el socket.
4. Dentro de un servicio, todos los sockets `requires` deberían encontrarse "enlazados" a algunos socket `provides` de otros componentes del despliegue. Describimos este mecanismo para especificar estos enlaces en la parte 2.
5. El objeto `export` contiene un atributo por cada socket TCP/IP estándar al que deba asociarse este componente. Estos sockets no se "enlazarán" a ningún otro socket

perteneciente a los componentes del despliegue. Un uso típico de dicho socket es aceptar peticiones HTTP de cualquier lugar de internet.

Echemos un vistazo al fichero `balancer/config/default.js`:

```
01:   module.exports = {
02:           external : {
03:               web: 8000
04:           },
05:           provides : {
06:               routerport : "8001"
07:           }
08:   };
```

De esta especificación averiguamos que el `balancer` no necesita ninguna URL para conectar con nadie. Al contrario, proporciona un socket `zmq` cuyo puerto de enlace por defecto es `8001`. Además, ofrece un puerto TCP/IP externo (`web`).

Para acceder a su configuración, el componenete tiene que `require` el paquete "`config`", devolviendo un objeto que se ajuste al mismo formato que el exportado por el anterior fichero `default.js`. En consecuencia, a modo de ejemplo, en el caso del `balancer`, las siguientes instrucciones …

```
01:   var cfg = require('config')
02:   cfg.provides.routerport
```

…se referirían al puerto que debe ser asociado por un socket `zmq` dentro del componente `balancer`. En este caso de ejemplo, salvo alteración por parte de quien efectúe el despliegue, este puerto será el 8001.

El archivo `default.js` para el `frontend` es

```
01:   module.exports = {
02:       requires: {
03:         balancerurl: "tcp://localhost:8001"
04:       },
05:       parameter: {
06:         maxload: 4
07:       }
08:   };
```

En este caso observamos que este componente no ofrece ningún *endpoint* al que otros componentes puedan enlazar. Sin embargo declara tener un puerto que necesita ser asociado con el URL adecuado. El valor por defecto que proporcionamos posee utilidad únicamente para testeo. En un despliegue real se debería asociar adecuadamente.

Además, el componente `frontend` declara un dato como parámetro, `maxload`, con `4` como valor por defecto. El parámetro `maxload`, como puedes deducir de la lectura del código del `frontend`, configura el número máximo de solicitudes web simultáneas que puede manejar el `frontend`.

2.1.2 Empaquetamiento de componentes

El paso final para la especificación de un componente en este ejercicio práctico consiste en la especificación de alguna forma concreta de empaquetamiento para el despliegue.

Nuestro mecanismo de empaquetamiento es una imagen `docker`. Una vez hayamos terminado con la codificación de un componente, debemos crear una imagen que encapsule todo el entorno del componente, junto con los materiales (código) que necesite para funcionar en este entorno.

Para conseguirlo, además del código del componente, también necesitamos especificar y construir una imagen `docker` a medida, de forma que podamos realmente lanzar instancias suyas. A tal fin necesitamos suministrar un `Dockerfile` dentro del directorio de cada componente, y ejecutar la siguiente orden dentro de dicho directorio

```
> docker build –t <nombre de la imagen del componente> .
```

La orden anterior produce como resultado la imagen `docker` que empaqueta el componente y lo deja listo para el despliegue.

Un vistazo rápido a cualquiera de nuestros dos componentes de ejemplo (*balancer* o *frontend*) descubre que cada uno de ellos, tal y como se esperaba, tiene un `Dockerfile`. Mirando sus Dockerfiles, descubrimos que… ¡son idénticos!

```
01:   FROM tsir/baselab3
02:
```

Lo único que declaran es su imagen base, que coincide con la que hemos creado antes para esta práctica.

¿Qué está ocurriendo?

La explicación se encuentra en las directivas `ONBUILD` que presentamos al discutir el `Dockerfile` para `tsrImage`. Las instrucciones cubiertas for dicha directiva se ejecutan justo ahora, cuando se construye la imagen de un componente, instalando el software del componente y sus dependencias dentro de la imagen que se está construyendo.

Continuemos y construyamos las imágenes correspondientes a nuestros dos componentes de ejemplo.

```
> cd Components/balancer
> docker build –t tsir/balancer .
```

y

```
> cd Components/frontend
> docker build –t tsir/frontend .
```

Mientras ejecutamos las órdenes anteriores podemos apreciar las órdenes activadas por `ONBUILD`-a medida que se construye las imágenes de cada uno de los componentes de nuestro ejemplo.

2.1.3 Preparación del cliente, broker y worker de la práctica 2

Ahora que sabemos cómo crear y empaquetar componentes, dejándolos preparados para lanzar instancias, preparemos los componentes necesarios para nuestros programas cliente, *broker* y *worker* de la práctica 2.

Creemos, dentro del directorio `Components`, los directorios `lab2client`, `lab2worker` y `lab2broker`. Dentro de cada uno de estos directorio debemos colocar todo lo que necesitemos para

a) Tener un componente, como hemos explicado antes
b) Poder generar una imagen `docker` (p.e., un `Dockerfile` adecuado)
    a. Pista: usa el mismo `Dockerfile` que el del componente *balancer*.
c) Conservar la configurabilidad que dichos componentes tenían en la práctica 2, pero adaptándolos a nuestro sistema de configuración. Es decir, los sockets deberán ser configurados automáticamente tras su especificación en el fichero `default.js`. Adicionalmente, las identidades de las instancias son automáticamente asignadas y están accesibles (ver más adelante), por lo que no hace falta preocuparse de asignarlas.

Tras conseguir lo anterior, deberíamos construir las imágenes para cada uno de estos componentes. Llamémosles `tsir/lab2client`, `tsir/lab2worker` y `tsir/lab2broker`, respectivamente.

Tras terminar, la orden

```
> docker images
```

Debería mostrarnos en su salida las siguientes imágenes

- `tsir/lab2broker`
- `tsir/lab2worker`
- `tsir/lab2client`
- `tsir/frontend`
- `tsir/balancer`
- `tsir/baselab3`
- `base/archlinux`

3 PARTE 2: DESPLIEGUE DE SERVICIOS

En la parte 1 hemos recorrido el proceso de especificación de componentes que podemos usar en la definición y despliegue de servicios. Hasta ahora no hemos explicitado cómo se especifica un servicio con tales componentes, y ése será nuestro objetivo en la parte 2: cómo definir un servicio y cómo especificar sus despliegues.

3.1 Definición de servicio

Un servicio es el resultado de ejecutar varios componentes interrelacionados, o, con mayor exactitud, varias instancias de componentes interrelacionados.

Las relaciones entre componentes se basan en funcionalidades que uno de ellos provee y que el otro necesita. El acceso a las funcionalidades requeridas suele conseguirse con la intermediación de algún canal de comunicación entre los componentes (sus instancias…).

En nuestro case, los canales de comunicación se basan en los sockets `zmq`.

En la parte 1 vimos que un componente publica su oferta de funcionalidades a otros componentes añadiendo una entrada en el atributo "`provides`" de su fichero de configuración `default.js`. También vimos que, cuando un componente precisa la funcionalidad de otro componente, se añade una entrada en el atributo "`requires`" de su fichero `default.js`.

Una vez declaradas las ofertas y demandas de cada componente, ¿cómo relacionamos a los componentes que piden con aquellos que ofrecen? Para explicar cómo podemos (y elegimos para estas prácticas) hacerlo, vamos a presentar nuestra propuesta de especificación para la ***definición de servicio***.

Una definición de servicio consiste en un directorio, cuyo nombre es el nombre del servicio, conteniendo un único fichero `index.js`. El fichero `index.js` es un módulo `nodejs` que será referenciado en el `required` por los motores de despliegue.

El módulo *<nombre servicio>*`/index.js` debe seguir las siguientes reglas:

1. Exporta un objeto conteniendo tan sólo dos atributos: "`components`" y "`links`". El valor de cada atributo es también un objeto.
2. El objeto `components` contiene un atributo por cada componente que forma parte del servicio que se está definiendo.
    a. El nombre de atributo será el nombre por el que se identifica el componente dentro del servicio.
    b. El valor asociado con este atributo es un objeto que contiene dos atributos necesarios para realizar despliegues del componente como parte de los despliegues del servicio:
        i. `image`: Nombre de la imagen docker que empaqueta el componente. La necesitamos para lanzar el contenedor docker que ejecuta sus instancias.
        ii. `location`: Ruta que especifica el directorio que contiene al componente. El desplegador necesita acceder a la información del componente que aparece en su `config/default.js`. Éste es un método expeditivo de obtener esa información en esta práctica, aunque, estrictamente hablando, no es necesaria (*¿Por qué? ¿Cómo extraerías esa información de otro modo?*)
            1. Adicionalmente, esta ruta podría usarse para construir la imagen del componente *al vuelo* (ver sección de trabajo extra).
3. El objeto `links` describe cómo se relacionan los componentes entre sí dentro de este servicio. Para ello especifica un grafo dirigido entre los componentes a través de sus sockets. Se estructura como sigue:
    a. Cada uno de los componentes referidos en el servicio (en "`components`") que necesita un socket a otro componente tiene un atributo con su nombre de componente.
    b. El valor de este atributo es otro objeto que indica qué componente del servicio satisface esa necesidad.

i. El nombre del atributo es el nombre especificado en el "`requires`" de la descripción de configuración del componente sita en su fichero `default.js`.
ii. El valor del atributo es un par ordenado, representado como un array de dos componentes. El primer elemento es el nombre del componente destino que ofrece la funcionalidad requerida, el segundo elemento es el nombre de uno de los sockets proveidos por este componente destino (y especificado en el "`provides`" del `default.js` del componente destino).

Vamos a ver un ejemplo de esta estructura.

Debéis ahora copiar el directorio `Services` desde `lab3support` a vuestro depósito `Lab3`. Dentro del directorio `Services` podéis encontrar otro directorio de nombre `zmqWebServer`, con un fichero `index.js` dentro. Éste constituye un ejemplo de definición de servicio usando los ejemplos de componentes que hemos visto antes. Analicemos su contenido:

```
01:  module.exports = {
02:          components: {
03:              balancer: {
04:                 location: require.resolve("../../Components/balancer"),
05:                 image: "tsir/balancer"
06:              },
07:              frontend: {
08:                 location: require.resolve("../../Components/frontend"),
09:                 image: "tsir/frontend"
10:              }
11:          },
12:          links: {
13:              frontend: {
14:                 balancerurl: ["balancer", "routerport"]
15:              }
16:          }
17:  };
```

De acuerdo con la descripción de servicio que hemos hecho anteriormente, el servicio `zmqWebServer` usa dos componentes: `balancer` y `frontend`. El componente `balancer` puede encontrarse en "`../../Components/balancer`", (relativo al directorio de definición del servicio). El componente `frontend` se encuentra en "`../../Components/frontend`" (también relativo al directorio de definición del servicio).

Las imágenes de cada uno de los componentes se llaman `tsir/balancer` y `tsir/frontend`, respectivamente.

Destacamos el uso que se hace del método "**`require.resolve`**" para calcular realmente el valor de los atributos "`location`". Esto es una conveniencia que simplifica el manejo de caminos relativos desde el desplegador, ya que "**`require.resolve`**" calcula una ruta absoluta basada en la ruta relativa y el directorio que contiene la definición de servicio. Vosotros deberíais hacer uso de este método en vuestras definiciones de servicio.

En este servicio, el componente `frontend` requiere un socket, `balancerurl`. La definición del servicio especifica que este requisito sea satisfecho por el socket `routerport`, proporcionado

por el componente `balancer`. Esto viene especificado en el atributo `links` de la definición del servicio que estamos analizando.

3.2 Definiendo despliegues de servicios

Una definición de un servicio tan sólo describe la estructura del servicio. Para realizar un despliegue del servicio es preciso añadir alguna información extra, como:

- El número de instancias de cada componente que se deben lanzar.
- Los valores de los parámetros que deben diferir de los valores por defecto recogidos en el fichero `default.js` de cada componente.

El proceso de despliegue debe, además, encargarse de asignar puertos TCP/IP a los sockets provistos por los componentes involucrados en el servicio, pero esto debe realizarse automáticamente, sin que deban asignarse valores de puertos de forma explícita por parte del operario.

Así pues, necesitamos también alguna forma de representar un despliegue. Nuestra aproximación consiste en crear un directorio por despliegue dentro del directorio *Deployments*. Dentro de este diretorio creamos un modulo `index.js` que exporta el objeto de definición del despliegue. Este objeto debe contener los siguientes atributos:

1. `servicePath`: Ruta al directorio que contiene la definición del servicio del cual éste es un despliegue. Se aconseja el uso de **`require.resolve`** para la resolución de dicha ruta.
2. `config`: Objeto con un atributo por componente del servicio que estamos desplegando. El objeto asociado a cada componente es un objeto de configuración, similar al que dicho componente define en su fichero `default.js`. Sin embargo, este objeto tan sólo puede contener los atributos `external` y `parameter`.
3. `counts`: Un objeto con un atributo por componente. El valor de dicho atributo es el número de instancias que deben lanzarse para ese componente.

Veamos un ejemplo.

Copiad el directorio `Deployments` desde `lab3support` a vuestro depósito `Lab3` . Dento de `Deployments` podemos encontrar el directorio `d1_zmqWebServer`, que contiene un fichero `index.js`. Analicemos su contenido:

```
01:  module.exports = {
02:      servicePath: require.resolve("../../Services/zmqWebServer"),
03:      counts: {
04:         balancer: 1,
05:         frontend: 3
06:      },
07:      config: {
08:         balancer: {
09:            external: {
10:                web: 80
11:            }
12:         },
13:         frontend: {
14:            parameter: {
15:                maxload: 2
16:            }
```

```
17:          }
18:       }
19:   };
```

Esto constituye un despliegue para el servicio `zmqWebServer`. La descripción indica que se van a crear 3 instancias del `frontend` y tan sólo una del `balancer`.

Finalmente, el puerto `external.web` reservado para el `balancer` es el `80`. El parámetro `maxload` del `frontend` recibe un valor de `2`.

3.3 Desplegando un servicio

Ahora que sabemos cómo describir componentes, servicios y sus despliegues, ya podemos proponer un algoritmo para realizar dichos despliegues.

Hemos preparado un código simple de despliegue basado en nuestras especificaciones.

Para utilizar este código, debéis copiar `lab3support/Deployers` a vuestro depósito `Lab3`. Copiad también el fichero `lab3support/deployer.js` al depósito `Lab3`.

Ya estamos preparados para realizar nuestro primer despliegue de servicio. Vamos a desplegar el servicio `zmqWebServer` con la siguiente instrucción:

```
> node deployer.js Deployments/d1_zmqWebServer
```

Deberemos comprobar el éxito de este despliegue utilizando nuestro navegador, dirigiéndolo al puerto `80` (el puerto `external web`) del contenedor `balancer`. Para poder hacerlo, antes deberéis averiguar la dirección IP de dicho contenedor. Docker nos permite hacerlo así:

```
> docker inspect --format '{{.NetworkSettings.IPAddress}}' balancer
```

Para tirar abajo el despliegue, podemos usar la siguiente instrucción

```
> docker rm –f balancer frontend frontent_1 frontend_2
```

donde `balancer`, `frontend`, `frontend_1` y `frontend_2` son los nombres que el desplegador asigna a cada uno de los contenedores que ejecutan las instancias desplegadas de los componentes. El desplegador sigue una regla muy simple para generar dichos nombres. El nombre de cada instancia tiene dos partes: un prefijo con el nombre del componente, seguido de un sufijo que es el número de instancia de dicho componente, separados prefijo y sufijo por el carácter "_". Hay una excepción: la instancia `0` de cada componente pierde el "_0" final, quedándose tan sólo con el nombre del componente.

Este nombre dado a los contenedores de `docker` actúa como identificador de cada una de las instancias, y está disponible para el código de un componente a través de su configuración del siguiente modo,

```
require('config').instanceId
```

Una inspección del código del componente `frontend`, os hará ver que este identificador es el usado para asignar una identidad al socket *dealer* usado por el `frontend`. Esta aproximación asegura que cada instancia recibe un nombre único dentro de su despliegue, nombre que puede ser usado para asignar identidades a los sockets.

3.4 Ejercicio: desplegando el servicio de la práctica 2

Ya habéis definido los componentes desarrollados en la prácitca 2: el *cliente*, el *broker* y el *worker*. Para poder realizar un despliegue, hay que definir su servicio. Además, para desplegarlo, necesitáis crear una definición de despliegue.

El ejercicio consiste en:

1. Crear una definición de servicio llamada `lab2Service`, y dejarla en el directorio `Services`.
2. Crear una definición de despliegue de nombre `d1_lab2Service`, y dejarla dentro del directorio `Deployments`. Debéis asegurararos de que esta definición,
    a. Crea 3 instancias del cliente
    b. Crea 6 instancias del worker
    c. Sólo hay 1 instancia del broker
3. Crear un programa `deployerlab2.js` directamente bajo el directorio principal del depósito `Lab3`. Este programa debe aceptar los siguientes parámetros
    a. Número de instancias del cliente
    b. Número de instancias de worker.

    Su misión es desplegar el `lab2Service` sin necesidad de una definición explícita del despliegue, y con 1 broker y el número de instancias de cliente y worker especificadas en los parámetros anteriores.

4 PARTE 3: CONSTRUYENDO VARIOS SERVICIOS GENERADORES DE IMÁGENES

Al verificar el comportamiento del servicio `zmqWebServer` desplegándolo con el despliegue `d1_zmqWebServer`, nos damos cuenta de que hemos construido un servicio web bastante monótono: siempre devuelve el personaje ése de *vendetta*.

Vuestro servicio `lab2service` tampoco ofrece un comportamiento mucho más interesante…

En esta parte de la práctica, vamos a combinar las arquitecturas de ambos servicios para obtener una solución escalable para un servicio de generación de etiquetas.

En nuestra arquitectura, el `balancer` genérico se encarga de la traducción de las peticiones HTTP a mensajes `zmq`, enviando el resultado a alguna de las instancias disponibles del `frontend`. El componente `balancer` nos sirve sin tener que realizar ningún cambio sobre él.

El trabajo principal de un componente, como el `frontend` proporcionado como ejemplo, es el procesamiento de la URL de petición. Basándose en esa URL, el `frontend` debe decider qué función realizar, posíblemente contactando con otros componentes capaces de realizarla. Ahora mismo, la única función que es capaz de realizar es servir un par de ficheros: el formulario que un usuario debe usar para pedir una etiqueta con un cierto estilo, y el fichero de estilo tsr.css usado por el navegador para formatear adecuadamente el formulario servido.

Sin embargo, el código en `frontend` no es capaz de procesar la petición de la etiqueta cuando el usuario rellena el formulario y lo envía. Para procesar esa petición debe ser capaz de llamar a otro componente.

4.1 El componente *LabelBuilder*

Dado que suponemos que la labor de generación de una etiqueta consume abundantes recursos (CPU, memoria), queremos evitar saturar nuestro componente `frontend` con dichas tareas, prefiriendo en su lugar que sea otro componente quien se encargue de ello. Así, para adaptarse a la carga esperada, podemos instanciar este componente especializado tantas veces como nos haga falta.

Debéis definir e implementar dicho componente, al cual vamos a llamar `LabelBuilder`. Por lo tanto, deberéis crear el directorio correspondiente (`LabelBuilder`) dentro del directorio `Components`. Este directorio deberá contener la implementación del componente. Para facilitaros la implementación de la parte de generación de etiquetas hemos preparado un paquete `nodejs` que podéis importar (`require`) desde la implementación de `LabelBuilder`. Este paquete está disponible en `lab3support/Support/imageBuilder`.

Para encaminar las peticiones desde las instancias de `frontend` a las instancias de `LabelBuilder`, vamos a usar el componente `broker` que ya habéis preparado.

Comparando esta tarea con la que realizasteis al final de la practica2, vemos que el `frontend` juega el papel del cliente del `broker` en dicha práctica, mientras que el `LabelBuilder` juega el papel del `worker`, recibiendo peticiones desde las instancias del `frontend`.

4.2 Nuevo frontend

Con el fin de hacer que el `frontend` sea capaz de enviar peticiones al `LabelBuilder` a través del `broker`, necesitamos modificarlo para adaptarlo, definiendo un socket adicional para encargarse de esa conversación con el `broker`. Además deberá manejar el envío del formulario desde el browser para extraer los parámetros de generación de las etiquetas que deberá pasar al `LabelBuilder` a través del `broker`.

Así pues, definiremos e implementaremos un Nuevo frontend, basándolo en el código del de ejemplo. El Nuevo frontend deberá satisfacer los siguientes requerimientos,

1. No debe limitar la concurrencia de peticiones al LabelBuilder. Es decir, debe permitir lanzar peticiones concurrentes al `LabelBuilder` a través del `broker`. Ello implica la correcta elección del tipo de socket a usar para esta tarea.
2. La composisción de `default.js` debe permitir su correcta configuración cuando se despliegue.

Este componente recibirá el nombre de `frontendPlus`, y deberéis crear el directorio con sus contenidos dentro del directorio `Components`.

4.3 El servicio webLabel

Con los components que tenemos ya definidos estamos preparados para definer el servicio de genración de etquetas. Vamos a llamarlo ***webLabelService***.

Como de costumbre, creamos un directorio bajo el directorio `Services` con el nombre `webLabelService`.

El diagrama siguiente debe servir de ayuda para obtener la definición del servicio:

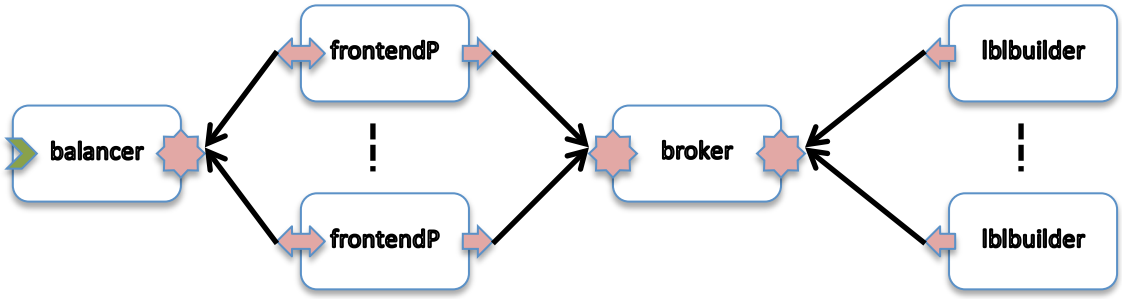

Figure 1: Arquitectura de webLabelService

Una vez definido el servicio, debéis testearlo con un par de despliegues que debéis definir: `d1_webLabelService` y `d2_webLabelService`. Estos despliegues deben presentar las siguientes características:

- `d1_webLabelService` solo crea una instancia de cada componente. El parámetro `maxload` del `frontendPlus` es 4.
- `d2_webLabelService`, sin embargo, tiene 2 instances for the `frontendPlus`, pero 4 instancias del `LabelBuilder`. El `maxload` para el `frontendPlus` se reduce a tan sólo 1.

5 EXTRAS

En este apartado se presenta algunas actividades opcionales. Deberías plantearos cómo resolverlas, y, en caso de éxito, contribuirán a mejorar vuestra calificación.

5.1 Alterando la referencia a componentes desde la definición de servicio

Dentro del esquema presentado en esta práctica, los components se empaquetan como imágenes `docker`, conteniendo toda la information sobre ellos, además del entorno de ejecución que precisan. Sin embargo, cuando nos referimos a un componente desde la definición de un servicio, se aporta tanto la imagen docker como la ruta a la implementación del componente, lo que parece redundante.

La única justificación razonable para tener que aportar la localización de la implementación del componente es el poder generar su imagen *al vuelo*. Por otra parte, conocer cuál es la imagen asociada a un componente nos debe permitir obtener toda la información que necesitamos sobre él. En particular, el contenido de su `default.js` que describe su configurabilidad.

Proponemos los siguientes ejercicios:

1. **Omitir la localización del componente (atributo `location`) de la definición de un servicio**. Cambiar de algún modo el motor de despliegue para hacer innecesaria la aportación de este campo. Para ello deberéis mostrar (e implantar) cómo obtener los contenidos del fichero `default.js` de un componente a partir de su imagen. Llamad `noLocationDeployer` al desplegador resultante, dentro del directorio `Deployers`.

2. **Omitir la imagen del componente (atributo `image`) de una definición de servicio**. Alternativamente, cambiar el motor de despliegue para construir la imagen del componente para lanzar sus instancias. Llamad al resultado `noImageDeployer`, y dejadlo en el directorio `Deployers`.

5.2 Permitir enlaces múltiples a sockets requeridos

En el transcurso de esta práctica, no se os puede haber escapado el detalle de que la semántica de nuestro modelo de construcción de servicios **no permite** que un componente que provee un socket sea instanciado más de una vez. Lo que significa que nuestro desplegador no puede manejar bien los despliegues que requieran más de una instancia para alguno de esos componentes.

Sin embargo, hay ocasiones donde tiene sentido proveer más de una instancia de ese tipo de componentes. En esos casos las aplicaciones cuyos sockets requeridos son enlazados con un socket proporcionado por un componente múltiplemente instanciado deben tener acceso a todos los puertos instanciados del componente enlazado.

Realizad una modificación del motor de despliegue que sea capaz de conseguir este efecto. Llamadlo `multiDeployer`.

6 ENTREGA

En la fecha de entrega (Jueves, 15 de Enero de 2015), el depósito canónico de cada equipo (Lab3) debe contener todo el material requerido:

1. Informe, describiendo el trabajo realizado, incluyendo la identificación de los miembros del equipo, y los comentarios de código que se estimen necesarios.
2. Ficheros de código.

6.1 Detalles del material a incluir

El depósito **Git** canónico, **Lab3**, debe contener el siguiente material:

1. **baseimages** (tsrImage)
2. **Components** (balancer, frontend, frontendPlus, lab2client, lab2broker, lab2worker, LabelBuilder).
3. **Services** (zmqWebServer, lab2Service, webLabelService)
4. **Deployments** (d1_zmqWebServer, d1_lab2Service, d1_webLabelService, d2_webLabelService)
5. **Deployers** (basicdeployer, <noLocationDeployer>, <noInstanceDeployer>, <multiDeployer>)
6. **deployerlab2.js**.
7. ***memoriaTSR3.pdf***

A título de guía, la memoria (***memoriaTSR3.pdf***) debe seguir la siguiente estructura:

- Portada:
  - Grupo de laboratorio
  - Componentes del equipo (apellido, nombre, email)
  - Indice
- Para cada parte:
  - Nombre y resumen de las tareas
  - Problemas relevantes encontrados y su resolución
    - Código relevante
- Detalles específicos de cada parte a destacar
- Comentarios globales
- Bibliografía usada