TSR - PRÁCTICA 3
CURSO 2016/17

DESPLIEGUE

Las modificaciones sobre la versión publicada el 28/11 consisten en:

- La línea `FROM fedora` en el Dockerfile de `fedora-node-devel` cambia a `FROM fedora:24`
- Se ha añadido una orden `docker build` al principio de la parte 2 para ilustrar cómo generar la imagen `tsr/fedora-node-devel`
- La última referencia a `WordPress` en el texto del último anexo se ha cambiado por `misitio`
- Se ha añadido una aclaración en el segundo "requisito para la realización de esta práctica" relacionado con la posibilidad del usuario de interactuar con el demonio docker.
- Los cambios de peso se han aplicado al apartado 3.4, aunque afectan también al 3.5
- Ha cambiado ligeramente el código de partida de `Lab3myTcpProxyA.js`

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"

En este documento, la **parte 1** se refiere al manejo básico de Docker, herramienta que usaremos como base de nuestra estrategia de despliegue. Se limita a referenciar algunas actividades descritas en el seminario 4 como toma de contacto.

La **parte 2** se centra en una estructura cliente-broker-trabajador, modelando los 3 componentes y consiguiendo, al final, el despliegue de un servicio en el que los 3 intervengan. Es muy interesante la perspectiva de servicio conseguida mediante docker compose. Secundariamente se tratan algunos aspectos relacionados con la reutilización de imágenes en contenedores.

El tiempo necesario para estas 2 partes debería ser inferior a 1.5 sesiones, dado que la mayoría de las acciones están indicadas.

Por el contrario, la **parte 3** es más ambiciosa: partiendo de un servicio de generación de imágenes para clientes web, se pretende realizar varias modificaciones que enriquecen su operativa o que ilustran combinaciones interesantes. En este caso el alumno ya no encuentra acciones que repetir o esquemas que le orienten, tratándose de problemas de carácter más abierto.

Para esta práctica se necesita:

1. Los materiales accesibles en PoliformaT (`tsr_lab3_material.zip`) en el directorio correspondiente a la tercera práctica. Puedes comparar su contenido con la imagen del último anexo.
2. La máquina virtual asignada al principio de curso, que contiene ya una instalación de Docker y permite realizar los ejercicios planteados en esta práctica.
   - En caso de emplear una cuenta de usuario *normal*, que no cuenta con privilegios de administración, no será posible interactuar con el demonio docker. Para permitirlo se deberá ejecutar:

```
sudo usermod –aG docker $(whoami)
```

   …. que añade al usuario invocante al grupo docker. Deberás volver a iniciar sesión para que tenga efecto.

CONTENIDOS

0 INTRODUCCIÓN

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

1. 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.
2. Otro problema a abordar consiste en la **configuración** de cada una de las instancias a desplegar.
3. También destacamos la necesidad de especificar la **interrelación** entre los diferentes componentes de una aplicación distribuida, especialmente el enlace entre *endpoints*: cómo se pueden definir y resolver.
4. Por último, el servicio obtenido y desplegado debe ser capaz de **escalar** sus recursos en función de la demanda, aspecto que deberá ser considerado tanto en la especificación de la configuración de dicho despliegue, como en el mecanismo que ejecute un escalado estático (iniciar con un número de instancias de cada componente) reconstruyendo los enlaces adecuados.
    - La opción dinámica es más compleja porque requiere crear o destruir instancias bajo demanda, sin perjudicar la disponibilidad del servicio. Abordar esta funcionalidad requeriría más esfuerzo del que disponemos en estas prácticas.

Una gran parte de los conceptos que se ponen en juego en esta práctica tienen su base en el escenario descrito en el apartado 3 del tema Despliegue de Servicios, aunque también intervienen otros condicionantes prácticos que no pueden ser ignorados.

En este laboratorio exploramos formas de construir componentes, configurarlos, conectarlos y ejecutarlos para formar aplicaciones distribuidas escalables de una forma *razonablemente* sencilla. Para ello nos dotamos de tecnologías especializadas en este ámbito.

1. Nos enfrentamos al primer problema con la ayuda del *framework* Docker. Tal y como se ha estudiado en el seminario, Docker nos provee de herramientas para preparar de forma reproducible toda la pila software necesaria para la instanciación de un componente, de manera que todas sus dependencias software las tenga disponibles sin interferir en las dependencias de otros componentes.

   Además, esta pila software se deja preparada para que la ejecución de un componente (su instanciación) suceda dentro de un contenedor Linux. Los contenedores Linux permiten "aislar" la ejecución de tareas de modo que lo que sucede dentro de un contenedor no afecte a lo que sucede dentro de otro contenedor, contribuyendo de manera apreciable a la reproducibilidad de la instanciación de un componente.

2. Para resolver adecuadamente el segundo problema se necesita especificar con metadatos la configurabilidad de cada componente, de manera que sea inteligible por un programa que lleve a cabo las acciones oportunas. Dada nuestra elección de tecnología (**Docker**) deberemos entender cómo dar a conocer las dependencias para

que el *framework* de Docker las resuelva. Concretamente necesitaremos conocer cómo cumplimentar un ***Dockerfile*** y cómo referenciar a algunas de las informaciones contenidas en él.

3. Para abordar la tercera necesidad descrita procederemos incrementalmente.
    a. Inicialmente desplegaremos todos los componentes de un servicio manualmente, usando directamente comandos docker, usando directamente las informaciones de configuración necesarias como parámetros de las órdenes docker.
    b. Posteriormente automatizaremos esta actividad mediante el uso de la utilidad **docker-compose**, que establece una forma de especificar las relaciones entre componentes para desplegar una aplicación distribuida. Esto nos obligará a entender los fundamentos de este nuevo programa, y la especificación necesaria[1] para construir el fichero **docker-compose.yml** con la interrelación entre los componentes de nuestra aplicación distribuida.
4. Finalizando nuestra relación de soluciones a necesidades parciales, el escalado forma parte de la funcionalidad ofrecida por **docker-compose**, de manera que el aspecto instrumental es mucho más sencillo de abordar. Por ello, llegado este punto, la dificultad principal radica en la adecuación de los componentes de la aplicación distribuida para su escalado. Se desea que el escalado contribuya a mejorar el rendimiento del servicio implementado, lo que no puede ser reducido a un asunto únicamente aritmético.

Los ejercicios concretos que se exponen utilizan una combinación de aplicaciones comunes típicas en este contexto (servidores web basados en APACHE, piezas autónomas como HA-Proxy), de las que no tenemos control interno de su código, y aplicaciones realizadas por nosotros (código en NodeJS, con módulos) que podemos modificar para alcanzar nuestros objetivos. Este segundo caso permite explotar y entender mejor la tecnología elegida (**Docker**).

[1] Mediante el lenguaje de marcas YAML

1 PARTE 1: PRIMEROS PASOS CON DOCKER EN EL SEMINARIO 4

En el primer ejemplo de uso de Docker del **seminario 4** se describe el despliegue un servidor de web "pata negra". Debes realizar esa misma actividad en tu máquina virtual, comprobando desde un navegador en tu equipo de escritorio (no en la virtual) que puedes acceder al servicio, y que te devuelve una página de hipertexto.

El material necesario para la imagen anterior se encuentra en la carpeta `misitio`. Después debes incluir el componente mencionado en la guía de estudio del seminario 4:

Lab3myTcpProxyA.js *(nueva versión del 02/12)*

Código (Lab3myTcpProxyA.js)

```
var net = require('net');

var LOCAL_PORT = 80;
var LOCAL_IP = '127.0.0.1';
if (process.argv.length != 4) {
  console.error("The WordPress port and IP should be given as arguments!!");
  process.exit(1);
}
var REMOTE_PORT = process.argv[2];
var REMOTE_IP = process.argv[3];
var server = net.createServer(function (socket) {
  var serviceSocket = new net.Socket();
  serviceSocket.connect(parseInt(REMOTE_PORT), REMOTE_IP, function () {
    socket.on('data', function (msg) {
        serviceSocket.write(msg);
    });
    serviceSocket.on('data', function (data) {
      socket.write(data);
    });
  });
}).listen(LOCAL_PORT, LOCAL_IP);
console.log("TCP server accepting connection on port: " + LOCAL_PORT);
```

Debes modificar dicho código de manera que se cumplan las siguientes condiciones:

- Todo el material debe ponerse en un nuevo directorio **tcpproxy**.

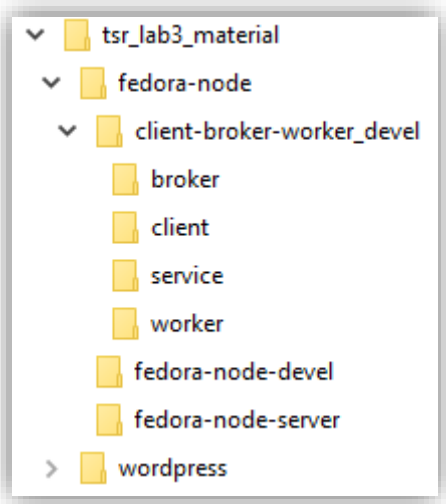
- Debe utilizar **tsr/fedora-node-devel** (comentada en el siguiente apartado) como imagen base
    - Observa que cada componente es realmente autónomo: éste se basa en fedora, mientras que el servidor con wordpress se basa en ubuntu.
- Atiende al puerto 80
- Las peticiones las reenvía al puerto de WordPress
- Se debe usar un único **docker-compose.yml** para todos los componentes.

Fíjate en que el resultado final debe ser un **docker-compose.yml** en el que se defina el despliegue de tres contenedores: uno para el proxy, otro para el servidor wordpress y otro para la base de datos de la que depende wordpress.

El proxy debe escuchar en el puerto 80, redirigiendo las peticiones al puerto que el servidor wordpress atiende. Debes configurar las credenciales que wordpress necesita para conectar con la base de datos. También necesitarás entender cómo acceder a las variables de entorno que inyecta Docker cuando se utiliza la directiva `link` en **docker-compose.yml**.

2 PARTE 2: REEDITANDO CLIENT-BROKER-WORKER, SU CONFIGURACIÓN Y ALTERNATIVAS

En el archivo de tsr_lab3_material.zip que acompaña a esta práctica puedes observar la presencia de un directorio `fedora-node-devel`, cuya estructura y contenidos se relacionan en el anexo "**client-broker-worker_devel**". En el subdirectorio `fedora-node-devel` puedes observar la presencia de un `Dockerfile`. Fíjate en este extracto:




```
FROM fedora:24
# …
RUN dnf install -y nodejs
# …
RUN dnf install -y zeromq-devel
# …
RUN dnf install -y make
# …
RUN dnf install -y python gcc-c++
# …
RUN dnf install –y ImageMagick ImageMagick-devel
# ...
RUN npm install zmq  md5 imagemagick
```

En él se muestra que, partiendo de una distribución Fedora, se añade el soporte para NodeJS y para ZMQ. Esto constituye el mínimo para la ejecución de muchos de los componentes que hemos desarrollado en la asignatura.

Construimos la imagen `tsr/fedora-node-devel` mediante la orden:

```
docker build –t tsr/fedora-node-devel .
```

Toda imagen o componente que dependa de éste tendrá capacidad para ejecutar aplicaciones NodeJS utilizando la biblioteca ZMQ.

En el directorio `client-broker-worker_devel` aparece el código y el `Dockerfile` de cada uno de los 3 componentes (client, broker y worker) de la aplicación. Comentamos el del cliente:

```
FROM tsr/fedora-node-devel
COPY ./client.js client.js
# We assume that each worker is linked to the broker
# container.
CMD node client $BROKER_URL
```

1. La orden COPY supone que el código del cliente se encuentra en este mismo directorio, y lo copia al contenedor que se está utilizando para construir la imagen.
2. Cuando se instancie un componente (lanzando un contenedor con la imagen del componente), se le pasará un valor como parámetro a través de una variable de entorno. Es una buena idea parametrizar el código a la espera de que *alguien* sustituya esas indicaciones por sus valores, en lugar de colocar valores fijos.
   - Para entender el mecanismo, debes estudiar la documentación de Docker relacionada con el entorno (*ENVIRONMENT*).
   - Recuerda que a esto lo denominamos "**inyección de dependencias**"

El worker admite una explicación análoga, y el `Dockerfile` del broker solo destaca por su orden **EXPOSE 8000 8001**, que está relacionada con las dependencias de los otros dos componentes.

Generar cada uno de los componentes no debe tener complicación; lo interesante es cómo interrelacionarlos: usando `docker-compose` y especificando una configuración en `docker-compose.yml` (ver directorio `service`).

```
version: '2'
services:
  wor:
    image: worker
    build: ../worker/
    links:
     - bro
    environment:
```

```
    - BROKER_URL=tcp://bro:8001
cli:
  image: client
  build: ../client/
  links:
   - bro
  environment:
   - BROKER_URL=tcp://bro:8000
bro:
  image: broker
  build: ../broker/
```

En el directorio `service/`, basta con ejecutar **docker-compose up** para tener la aplicación distribuida en marcha, incluyendo los pasos intermedios como la inyección de dependencias.

```
[root@TSR294 devel]# docker-compose up
Creating devel_bro_1
Creating devel_cli_1
Creating devel_wor_1
Attaching to devel_bro_1, devel_cli_1, devel_wor_1
cli_1 | Broker URL is tcp://172.17.0.88:8000.
cli_1 | Request: 16
cli_1 | Request: 7
cli_1 | Request: 74
cli_1 | Request: 92
cli_1 | Request: 72
cli_1 | Request: 22
cli_1 | Request: 70
cli_1 | Request: 50
cli_1 | Request: 16
cli_1 | Request: 13
cli_1 | Returned value: 32
cli_1 | Returned value: 14
cli_1 | Returned value: 148
cli_1 | Returned value: 184
cli_1 | Returned value: 144
cli_1 | Returned value: 44
cli_1 | Returned value: 140
cli_1 | Returned value: 100
cli_1 | Returned value: 32
wor_1 | Worker: The broker URL is tcp://172.17.0.88:8001
wor_1 | Reply: 32
wor_1 | Reply: 14
wor_1 | Reply: 148
wor_1 | Reply: 184
wor_1 | Reply: 144
wor_1 | Reply: 44
wor_1 | Reply: 140
wor_1 | Reply: 100
wor_1 | Reply: 32
wor_1 | Reply: 26
cli_1 | Returned value: 26
devel_cli_1 exited with code 3
^CGracefully stopping... (press Ctrl+C again to force)
Stopping devel_wor_1 ... done
Stopping devel_bro_1 ... done

Aborting.
[root@TSR294 devel]# ^C
[root@TSR294 devel]#
```

Mientras la aplicación se encuentra en marcha, podemos modificar el número de workers activos con la orden **scale**, que establece el número de contenedores para un servicio. Como seguramente el único cliente[2] ya habrá terminado, el broker y los workers se habrán quedado en espera. Podemos detenerlos pulsando Ctrl+C. Alternativamente, si abrimos otro terminal y vamos a ese mismo directorio (es decir, allí donde esté el docker-compose.yml), con esta orden...:

```
docker-compose scale cli=10 wor=5
```

...generaremos 10 nuevos clientes y aumentaremos el número de workers a 5.

Recuerda que los clientes terminan tras unos envíos, por lo que los podemos ir reponiendo con **docker-compose scale cli=10**

Compose puede gestionar imágenes e instancias de contenedores, y suele llegar un momento en que necesitamos conocer o eliminar algunas de esas piezas. Las órdenes más útiles para esto son ps, rm y kill (consulta el manual).

2.1 Modificaciones para client-broker-worker_server

Es conveniente diferenciar los requisitos de desarrollo de los requisitos de explotación, de manera que la presencia de herramientas o bibliotecas de desarrollo no son deseables si no son imprescindibles. Sabemos que el software sobrante puede ser fuente de problemas, de manera que no queremos incorporar lo que no sea necesario en las imágenes de las que se derivan nuestros componentes.

[2] consulta su código

En el caso anterior podemos identificar la presencia del software **npm** para instalar automatizadamente bibliotecas necesarias para NodeJS. Esto acarrea la instalación de aplicaciones y bibliotecas adicionales de cierta envergadura. Por ejemplo, para poder compilar el código fuente del módulo "zmq" se ha necesitado instalar los paquetes "python", "gcc-c++" y "make" en la imagen que hemos construido. Además de la ocupación extra (más de 200MBs), esos paquetes no serán necesarios posteriormente. ¿Cómo nos deshacemos de ellos?, ¿podemos desinstalarlos desde los componentes client/broker/worker?

- En resumen: sí, podemos quitarlos con las operaciones propias de Fedora (`dnf uninstall python gcc-c++ make`), pero puede dejar muchos residuos en la instalación y no recuperará todo el exceso consumido.
- Una buena alternativa consiste en generar el directorio `node_modules` con los contenidos necesarios desde una imagen `fedora-node-devel`, tal y como se hace en el `Dockerfile` presentado más arriba. Una vez generado el `node_modules` lo capturamos copiándolo a nuestro directorio /tmp (ver más abajo). Una vez hecho esto, generaremos las imágenes para nuestros componentes, client, broker y worker, en otra imagen más básica que `fedora-node-devel`, no incluyendo ninguno de los elementos utilizados para el desarrollo. Para conseguir esa imagen más básica basta eliminar del `Dockerfile` de `fedora-node-devel` las líneas `RUN` que instalan los elementos de desarrollo y la que ejecuta npm. Para que nuestra aproximación funcione deberemos encontrar una forma de hacer que el directorio `node_modules` capturado anteriormente aparezca en los contenedores de nuestros componentes. Llamaremos "`tsr/node-base`" a esa segunda imagen más reducida en la que basaremos las imágenes de nuestros componentes.

En detalle:

1. Ejecutar una instancia de tsr/fedora-node-devel y dejarla en suspenso. Mediante "docker ps" obtendríamos el ID de ese contenedor. Asumamos que es cont-ID. Copiaremos al /tmp del anfitrión su directorio node_modules

```
docker cp cont-ID:node_modules /tmp/node_modules
```

2. Modificar el Dockerfile de los componentes para que dependan de tsr/node-base (en lugar de la tsr/fedora-node-devel original)
3. Y ahora decidimos cómo incorporarlo a las imágenes de client, broker y worker:
    - ¿Copiando desde el Dockerfile de cada uno?
    - ¿Compartiendo como un volumen? Busca en la documentación…
    - ¿Hay otras alternativas?[3]

En resumen, en el equipo de desarrollo creamos los requisitos necesarios para *adjuntarlos* al software que diseñamos como componentes del servicio.

La actividad a desarrollar en esta segunda parte consiste en:

1. Revisar y comprender el código de los componentes client, broker y worker. Estudiar sus dependencias. Por ejemplo, si modificáramos el código del broker para que sus

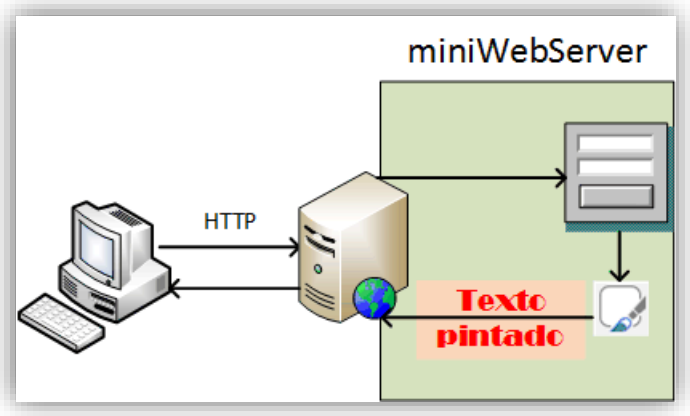
[3] Sí!!

sockets utilizaran los puertos 8010 y 8011 en lugar del 8000 y 8001 actuales...¿Habría que introducir algún cambio en los demás ficheros NodeJS, Dockerfile o docker-compose.yml? ¡Compruébalo!!

2. Realizar los tres pasos enumerados anteriormente para incluir el directorio "node_modules" de la imagen "tsr/fedora-node-devel" en la imagen que utilizarán los componentes client, broker y worker.
3. Regenerar las imágenes de esos tres componentes. Comprobar, mediante la orden "docker-compose up" (desde la carpeta "service") que los tres componentes pueden interactuar sin problemas en esta nueva versión.
4. Utilizando "docker images" y "docker history", explicar la diferencia de tamaño entre las imágenes "tsr/fedora-node-devel" y las nuevas imágenes de cliente, broker y worker.

3 PARTE 3: DESDE UN SERVIDOR WEB HASTA UNA MODESTA APLICACIÓN DISTRIBUIDA

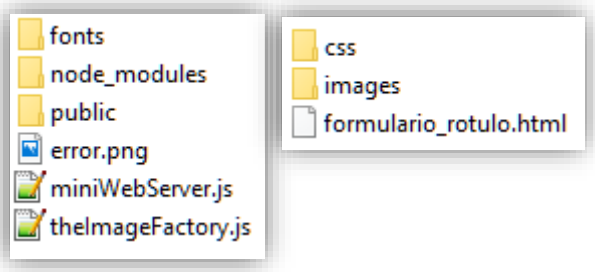
El propósito de esta parte es simular una adaptación progresiva de una pequeña aplicación, pasando por una descomposición en componentes autónomos y completando el despliegue tras cada fase. Aunque no se pretende generalizar ni marcar pautas, esperamos que tenga un carácter integrador y pedagógico en la construcción de aplicaciones distribuidas.



Disponemos de una aplicación NodeJS sobre web que, atendiendo a solicitudes de los usuarios, genera y devuelve rótulos (imágenes compuestas por texto). Para ello, la aplicación se ha construido básicamente sobre el módulo http y aprovecha una biblioteca gráfica para procesar las imágenes.

- Este último aspecto no es interesante para esta asignatura.

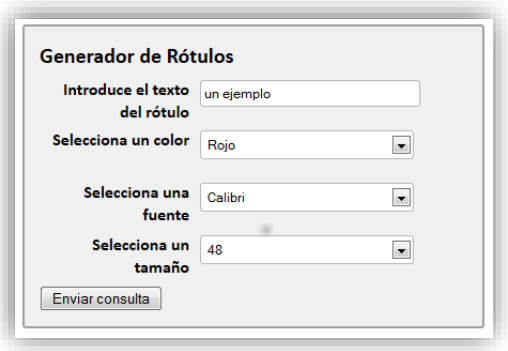
La aplicación (ya incluida en la carpeta "part3-server" dentro del ZIP "tsr_lab3_material.zip" mencionado en la introducción), no es, ciertamente, un prodigio. Para evitar el uso de promesas[4], el tratamiento de errores se ha simplificado (p.ej. incluso aunque haya un error, se devuelve una imagen). Ha prevalecido el carácter instructivo de las actividades a realizar sobre la ejemplaridad del código en el que se basa.



[4] Que sería la forma ortodoxa de abordarlo

La secuencia de funcionamiento de la aplicación viene a ser:

1. El usuario accede con un navegador al servidor, empleando el URL de un formulario.
    - Indirectamente, el navegador decide pedir también la hoja de estilos que viene referenciada en la página del formulario, y que el mismo servidor proporciona.
2. El usuario cumplimenta el formulario seleccionando los detalles que prefiere, y pulsa el botón de envío.
3. El servidor recibe la información y realiza el procesamiento (generar imagen) de los datos.
4. Cuando termina, devuelve la imagen generada.



Es conveniente familiarizarse con el código del servidor, los directorios en los que se organiza el material y su configuración.

En la secuencia que se acompaña pueden distinguirse **2 bloques**: los 3 primeros apartados representan acciones sobre un escenario heredado (***legacy***), en el que nos encontramos con una aplicación que no podemos modificar, pero cuyo servicio debemos mejorar. En situaciones reales hablaríamos de binarios que no podemos cambiar, o aplicaciones para las que nuestro contrato no permite su modificación. Toda nuestra mejora se limitaría a acciones externas a la aplicación.

En el segundo bloque, por el contrario, disponemos de la oportunidad y capacidad para intervenir en la aplicación, podemos diseñarla desde cero para dividirla en módulos que cumplan con los criterios de bajo acoplamiento y alta cohesión, o aplicar un rediseño en el código existente con criterios similares a los ya citados.

3.1 Despliegue inicial (miniWebServer.js)

El servidor de rótulos es un servidor web que se organiza como un solo componente (miniWebServer.js) que emplea la biblioteca imageBuilder.js, mediante la que crea rótulos que devuelve a los solicitantes.

- Requiere un Dockerfile y opciones especiales en la línea de órdenes para conectarlo con el puerto 80. Es importante comprobar que se satisfacen todas las dependencias, requisito que se cumple tomando como imagen de partida fedora-node-devel
- Dispone de una lista de URIs admitidas (recursos accesibles en el servidor), incluyendo:
    - Una página estática de formulario
    - Una hoja de estilo
    - Una función para el procesamiento del URI `/process`
        - También tiene otros requisitos, ya cubiertos, como la existencia de un directorio con imágenes

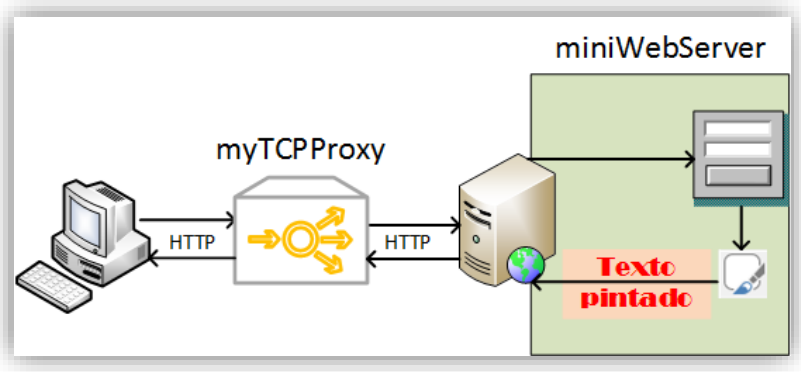
Se propone poner en funcionamiento el sistema en la máquina virtual, y, una vez comprobado, intentar hacerlo sobre un contenedor, ejecutando las instrucciones en modo interactivo para recopilarlas al final en un Dockerfile a medida.

Recuerda que, como administrador, puedes instalar los requisitos que sean necesarios; en cualquier caso, el Dockerfile debe ajustarse adecuadamente.

La solución a esta primera tarea ya se ofrece con un Dockerfile en la carpeta "part3-server". Sería interesante trasladar ese fichero (antes de consultarlo) a otro directorio para tratar de resolver este despliegue inicial sin ayudas.

3.2 Variación: añadir myTCPProxyA

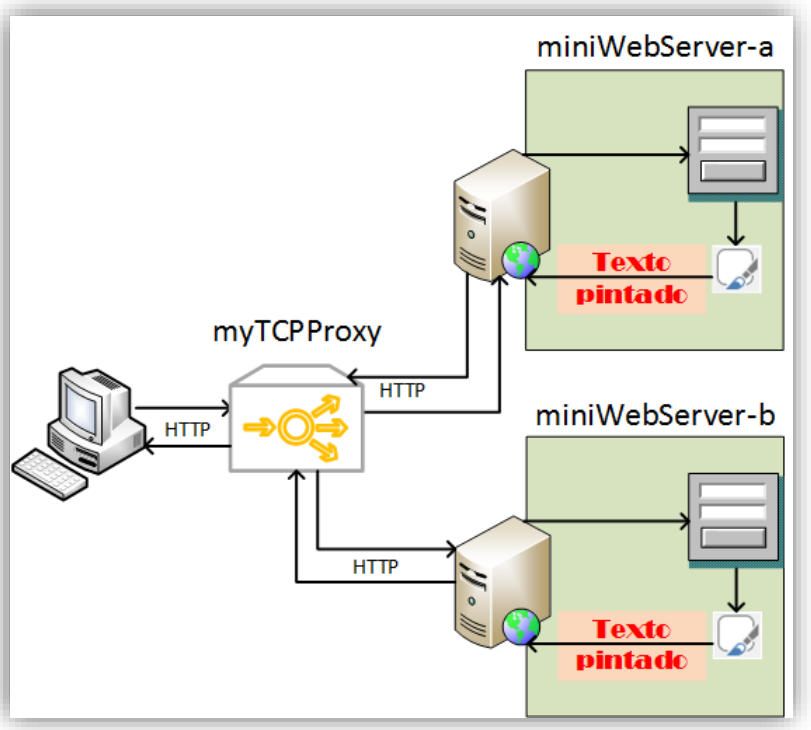
(El mismo que el del final de la parte 1). Se propone anteponer un myTCPProxy que intermediará entre el navegador y miniWebServer. Para ello el componente myTCPProxyA "heredará" el puerto externo anteriormente utilizado por miniWebServer, de manera que los clientes no puedan observar cambio alguno.



Este nuevo caso requerirá un despliegue conjunto mediante docker-compose.

3.3 Servidor bicéfalo

Colocar 2 instancias de miniWebServer, de manera que myTCPProxyA pueda repartir las peticiones entre ellos. El criterio de selección será el más sencillo: alternar rotatoriamente (Round-Robin) entre ambos.



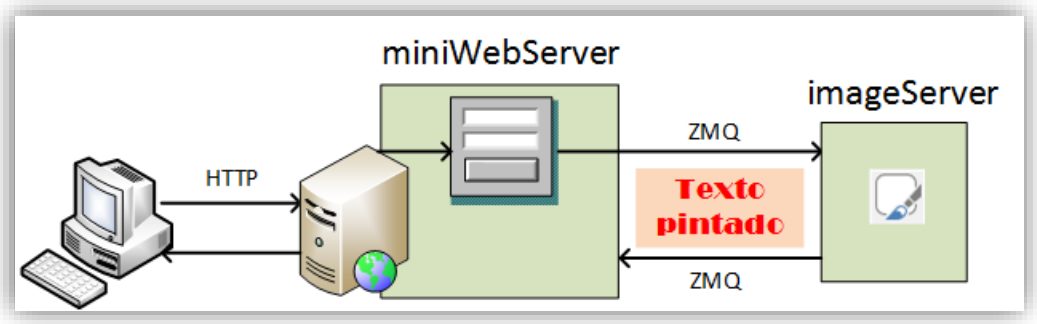
- No hay posibilidad sencilla de decidir el destino dependiendo del contenido de la petición, porque desde TCP no se entienden los mensajes HTTP (p.ej. el URL): es un desalineamiento de protocolos.
- Es sencillo, pero poco útil, establecer

estrategias de reparto que dependan de la información disponible en el nivel TCP, como la procedencia de la petición.

- Sería más complicado, pero muy útil, tomar criterios que dependan del estado de los trabajadores: disponibilidad (recordar heartbeat), carga actual …
    - Requiere comunicaciones adicionales y mantenimiento de cierto estado.
    - Seguro que ya estás familiarizad@ con estos conceptos y su implementación porque los hemos propuesto en la práctica anterior.
- Razona si es posible efectuar un despliegue con las siguientes restricciones:
    - Hay un número indeterminado de instancias de miniWebServer
    - Reparto del trabajo por Round-Robin
    - Sin el middleware ZMQ

3.4 El servidor de imágenes (imageServer)

Éste es el primer caso del segundo bloque, y, como tal, adopta como punto de partida la versión original del generador web de rótulos (miniWebServer), pero vamos a reestructurarlo de forma diferente.



Ahora conseguimos que el servidor web se haga cargo únicamente de la conexión HTTP, resolviendo solo las peticiones más sencillas de los clientes. Para producir la imagen solicitada, contacta con un proceso externo (imageServer) que ejecutará el código para generar el rótulo que teníamos en el apartado anterior.

En miniWebServer podemos identificar con cierta facilidad la parte dedicada al procesamiento de `/process`, y que se podría colocar en otro componente (**imageServer**, un trabajador) y que hemos de enganchar con el original (ahora renombrado a **miniWebServer2**). Debemos prestar atención a los siguientes aspectos y cuestiones:

- ¿Cómo comunicamos miniWebServer2 e imageServer?, ¿TCP/IP?, ¿HTTP?, ¿ZMQ? Es nuestra elección. Pueden encontrarse argumentos para usar el middleware de mensajería como vía de comunicación entre nuestros componentes (de hecho, se estudió en clase) y por ello elegimos ZMQ.
- Habiendo seleccionado ZMQ, ¿cuál es el modelo de comportamiento que necesitamos?: ¿REQ/REP? Pensad en las necesidades de concurrencia para elegir los sockets.
    - A estas alturas parece innecesario remarcar la similitud entre este escenario y el del client-broker-worker, pero también posee sus propias características diferenciadoras. Más detalles en el apartado 3.5

- Requiere que desarrollemos los programas `miniWebServer2` e `imageServer`, que creemos el `Dockerfile` de cada componente, y un `docker-compose.yml` para el despliegue de todo el sistema.

A la hora de **refactorizar** el código para convertir el miniWebServer actual en el nuevo miniWebServer2, debes resolver el problema del mantenimiento del contexto de la petición HTTP en miniWebServer2 durante el tiempo en que espere una respuesta de imageServer.

- En NodeJS, el contexto de una petición HTTP se mantiene en los parámetros (`req, res`) que recibe el servidor; es decir, se guardan en la clausura que NodeJS crea con ese manejador tras recibir la soliitud HTTP.

Dentro de miniWebServer, la producción del rótulo se realiza síncronamente con la ejecución de la función que trata la petición HTTP (el manejador invoca directamente la función que produce el rótulo), evitando la necesidad de mantener el contexto HTTP.

Sin embargo, en miniWebServer2 debemos resolver el caso asíncrono en el que miniWebServer2 envía un mensaje a imageServer, y que será resuelto más adelante, con la llegada a miniWebServer2 de un mensaje procedente de imageServer.

- En ese momento miniWebServer2 debe relacionar la llegada de la respuesta con el contexto original HTTP, de manera que pueda aprovechar el objeto `res` para enviar una respuesta al navegador.

En cierto modo miniWebServer2 se comporta como un broker especializado, recibiendo peticiones cliente de un navegador web (comunicaciones HTTP) y pasando las tareas encargadas (producción de rótulos) al trabajador imageServer.

- La principal diferencia es el uso de HTTP en lugar de ZMQ entre cliente y miniWebServer2, que obliga a mantener no solo el estado de la conexión TCP/IP representado por el socket, sino todo el estado adicional que HTTP pueda añadir.

Dado que ese **contexto** no puede ser transmitido en un mensaje[5], debe ser "almacenado" localmente. Una forma de hacerlo, ligando el contexto al mensaje, consiste en seleccionar una ubicación en un diccionario[6], colocar en ella una copia del contexto y enviar en el mensaje una clave que permita recuperar esa copia posteriormente.

- Simplificaremos usando un vector circular con un tamaño (maxSlots) arbitrario pero limitado. En cada petición de servicio a imageServer se adjuntará un identificador que referencie a la posición en la que miniWebServer2 ha guardado el objeto respuesta correspondiente.

[5] Una parte del contexto no son datos
[6] Un vector sería un caso concreto

Debes tener un cuidado especial a la hora de seleccionar número y tipo de sockets entre servidor y trabajador (miniWebServer e imageServer). También quién hace `connect()` y `bind()`.

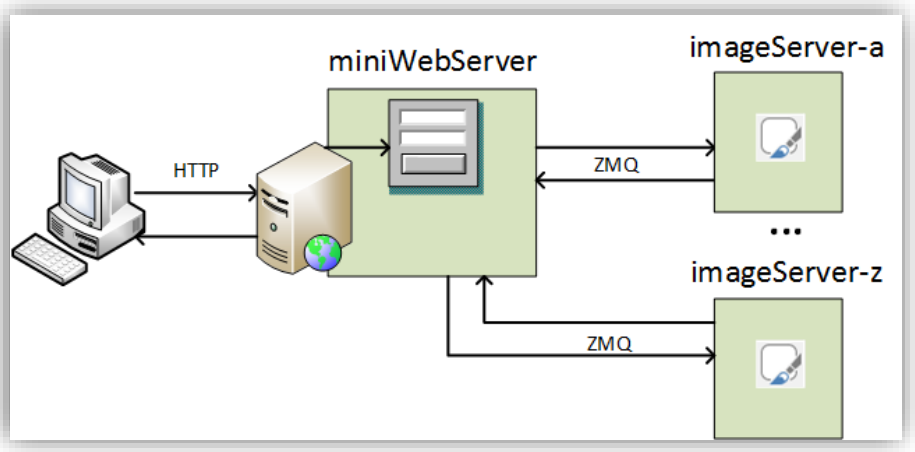
- Si piensas en canales unidireccionales, seguramente necesites 2. Si son bidireccionales, puede bastarte 1. Piensa que más tarde deberás generalizarlo a múltiples imageServer.
- El modelo síncrono o no del socket ZMQ influye en el rendimiento, y puede también provocar que la solución no sea correcta.
- Observa que miniWebServer2 no actúa exactamente como los brokers que conoces: el protocolo con el cliente es HTTP, por lo que miniWebServer2 no encaja como ROUTER-DEALER ni como ROUTER-ROUTER.
- Los parámetros de la conexión deberán ser modificables (argv) para poder insertarse en el despliegue.

El código básico relevante del nuevo **miniWebServer2** incluirá:

```
// Define ZMQ socket
var so    = zmq.socket('XXX');
// Define additional ZMQ sockets if needed
var so2   = zmq.socket('YYY');

// We must store pending HTTP responses
var responses = [];
// Current amount of processed responses (modulus maxSlots)
var numResponses = 0;

// Maximum number of slots in the "responses" array.
const maxSlots = 200;
// You must choose appropriate initialization operation: bind or connect
// argv[] containing protocol://transport:port details

so.bindSync(argv[2]); // it depends… connect or bind?
so2.connect(argv[3]); // it depends… connect or bind?

…

http.createServer(function(req, res) {

…

  } else {
          if (action == '/process') {
                var form = request.query;
                // Keep the HTTP response object in the "responses" array.
                responses[ numResponses ] = res;
                // Send as a first segment the slot number where the response is kept.
                // The second segment is the form to be processed by workers.
                so.send( [ numResponses, JSON.stringify(form)] );
                numResponses = (numResponses % maxSlots) +1;
        } else { // incorrect url
                res.writeHead(404, {'Content-Type': 'text/plain'});
                res.end("404 Not found");
          }
  };
}).listen(PORT,ADDR);
```

3.5 Variación: múltiples imageServer

Bajo la suposición de que las tareas de construcción de rótulos representan un elevado coste computacional, y dado que mayoritariamente son independientes entre sí, una forma de incrementar en rendimiento y/o la disponibilidad se basa en la disposición de réplicas.



A la hora de seleccionar el mecanismo usado por miniWebServer2 para repartir las peticiones entre los trabajadores, debemos exigirle flexibilidad para poder aplicar diferentes criterios de reparto de esa carga.

- ¿El esquema de esta imagen nos proporciona esa flexibilidad?
  - En caso afirmativo, ¿cuáles serían los sockets ZMQ adecuados?
- ¿Se te ocurre alguna forma de reaprovechar alguno de los componentes usados en la parte 2?
  - Como se ha apuntado anteriormente, hay similitudes con esos patrones tan familiares usados en la práctica 2 y en la segunda parte de esta misma práctica 3.
  - De nuevo, ¿cuáles serían los tipos de socket más adecuados para miniWebServer e imageServer? Puede que coincide con tu elección para el apartado 3.4, pero deberás comprobarlo.

Esto dará lugar a una carpeta **miniWebServer3** y, por coherencia, otra **imageServer3**, con sus códigos y sus Dockerfiles. También deberás escribir un nuevo `docker-compose.yml`

Es necesario que compruebes si puedes escalar a 4 ó 5 instancias de imageServer. No necesitas implementar cómo elegir a quién enviar una petición, pero necesitas identificar al menos qué implicaciones tendría un requisito como "enviar al imageServer menos ocupado". En el siguiente apartado leerás otras propuestas de reflexión.

4 CUESTIONES

Hemos experimentado con contenedores, pero no hemos proporcionado acceso a ellos desde el exterior, por lo que la primera cuestión es: ¿cómo puedo acceder a la página de bienvenida de cualquiera de los servidores web "*contenerizados*" desde un navegador de otro ordenador?.

Los aspectos relativos al tamaño de las imágenes pueden ser preocupantes en los casos en que el número de instancias (contenedores) derivados de esa imagen sea pequeño; por ello se realiza

un esfuerzo extra en disminuir dicho tamaño con varias estrategias (como evitar elementos superfluos). ¿Puedes apuntar otras?, ¿podrías "optimizar" estas imágenes?.

Sabemos que Docker en modo Swarm permitiría que los componentes que hemos manejado en esta práctica se puedan alojar en equipos distintos. ¿Serías capaz de indicar el mínimo despliegue de alguno de estos sistemas (elige el que prefieras) en 2 máquinas mediante Swarm?

Si los imageServer tuvieran una interfaz HTTP, y miniWebServer actuara como un repartidor, podría redirigir[7] cada petición que le llegue al imageServer adecuado. P.ej.,

- los rótulos que comiencen por [a-g] van a imageServer1,
- los que comiencen por [h-p] a imageServer2
- los que comiencen por [r-z] a imageServer3
- y el resto a imageServer4

En estas condiciones no se requiere que miniWebServer se comunique internamente con ellos, y podemos interpretar esta estrategia como un simulacro de *sharding*. Se necesita que miniWebServer incorpore (X) una parte de la información procedente del formulario. Discute qué posibilidades hay y sus implicaciones.

[7]
```
res.writeHead(301,
  {Location: 'http://'+imageServerIP+':'+imageServerPort+'/'+X});
res.end();
```

5 ANEXO: CLIENT-BROKER-WORKER_DEVEL

5.1 Directorio fedora-node-devel

5.1.1 Dockerfile

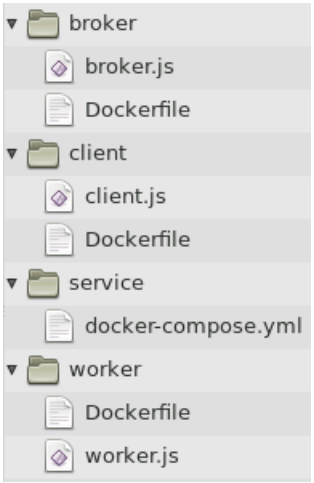
```
# Take the latest Fedora distribution as a base.
# Currently (November 22, 2016), it is Fedora 24.
FROM fedora:24
# Install the latest Node.js on that distribution.
# It is Node.js v4.6.1.
# We may use the dnf Fedora command to this end.
RUN dnf install -y nodejs
# Next step: Install the zeromq library.
# Its package is called "zeromq-devel" in the Fedora
# distribution.
RUN dnf install -y zeromq-devel
# We still need the standard "make" command in order
# to run "npm".
RUN dnf install -y make
# There are other "tools" needed for compiling the
# "zmq" module. They are: python and gcc-c++
RUN dnf install -y python gcc-c++
# Install the ImageMagick library. It is needed for
# developing the server described in Section 3.
RUN dnf install -y ImageMagick ImageMagick-devel
# Finally, let us run the npm command for installing
# our nodejs modules.
RUN npm install zmq md5 imagemagick
```

5.1.2 Acciones

```
docker build –rm –t tsr/fedora-node-devel .
```

5.2 Directorio client-broker-worker_devel

Compuesto por 3 directorios (`client`, `broker` y `worker`) con una aplicación NodeJS que implementa un componente configurado por un `Dockerfile` sencillo, un par de scripts para tareas típicas, y un directorio `service` con un archivo `docker-compose.yml` para automatizar el despliegue.



Sus contenidos han sido objeto de descripción en los apartados 5.1 y 6.4 del seminario 4.

6 ANEXO: IMAGESERVER

En este anexo no se incluye el formulario, la hoja de estilo ni los directorios auxiliares; únicamente el código de los dos archivos NodeJS de la versión inicial de la aplicación, así como el Dockerfile necesario para generar el servidor de imágenes.

6.1 miniWebServer.js

```
//
// miniWebServer.js
//
```

```
// Common section
var fs    = require('fs');
var path  = require('path');

// Web section
var http  = require('http');
var url   = require('url');
var PORT  = 8000;
var ADDR  = '0.0.0.0';
var PREFIX = path.dirname(__filename) + "/public/";

// Image section
var image_basedir = path.dirname(__filename) + "/fonts/";
var Rotulo = require('./theImageFactory');

//{valid_url:{path_from_PREFIX, mime}}
var contents={
  '/':{path:'formulario_rotulo.html', mime:'text/html'},
  '/index.html':{path:'formulario_rotulo.html', mime:'text/html'},
  '/formulario_rotulo.html':{path:'formulario_rotulo.html', mime:'text/html'},
  '/css/tsr.css':{path:'css/tsr.css', mime:'text/css'}
//  '/process':{path:'css/cover.css', mime:'text/css'}
};

http.createServer(function(req, res) {
  var request = url.parse(req.url, true);
  var action  = request.pathname;

  if (contents[action]){ // not undefined
    fs.readFile(PREFIX+contents[action].path,
      function (err, data) {
        res.writeHead(200, {'Content-Type': contents[action].mime,'Content-Length':
data.length});
        res.end(data);
    })
  } else
    {
    if (action == '/process') {
      var form = request.query;
      console.log(form);
      Rotulo.Label(form.texto, form.fuente, form.tam, form.color,
        function(err, rot){
          if (!err)  { // not undefined
              res.writeHead(200, {'Content-Type': 'image/png'});
              res.end(rot.image, 'binary');
          }
          else { // incorrect user data || internal error
            res.writeHead(404, {'Content-Type': 'text/plain'});
            res.end("404 Not found");
        }});
    }
    else { // incorrect url
      res.writeHead(404, {'Content-Type': 'text/plain'});
      res.end("404 Not found");
    }
  };
}).listen(PORT,ADDR);
```

6.2 theImageFactory.js (biblioteca)

```
//
// theImageFactory.js
//
var imgk  = require('imagemagick');
var md5   = require('md5');
var fs    = require('fs');
```

```
var path  = require('path');

var basedir = path.dirname(__filename) + "/fonts/";
var ERROR_IMG = fs.readFileSync(path.dirname(__filename) + "/error.png");
// Number-ordering for font type (16 selected)
var fonttypes = [
"ahronbd.ttf", "andlso.ttf",   "BOD_BLAR.TTF",
"BOOKOSB.TTF",  "BRITANIC.TTF", "BRLNSB.TTF",   "BRLNSDB.TTF",
"BROADW.TTF",   "PALSCRI.TTF",  "phagspab.ttf", "plantc.ttf",
"PRISTINA.TTF", "SketchFlow Print.ttf", "upcjb.ttf",    "upclb.ttf",
"verdanab.ttf"
];

// Sizes as reference for switch construction
var fontsizes = ["16", "24", "32", "40", "48"];

// Accessing colors by their names
var RGBcolors = {
    blanco:"#FFFFFF",   negro:"#000000",   rojo:"#FF0000",
    verde:"#00FF00",    azul:"#0000FF",    magenta:"#FF00FF",
    celeste:"#00C89B",  gris:"#BEBEBE",    salmon:"#FA8072",
    naranja:"#FFA500",  coral:"#FF7F50",   tomate:"#FF6347",
    rosa:"#FFC0CB",     marron:"#B03060",  orquidea:"#DA70D6"
};

function Image(text, font, size, colour) {
  this.text = text;
  this.font = font;
  this.siz = size;
  this.colour = colour;
  this.image = undefined;
  this.md5 = undefined;
  this.setMD5 = function(h) {this.md5 = h}
  this.setImage = function(i) {this.image = i}
  this.size = function() {return this.image.length}
  this.img = function() {return this.image}
  this.hash = function() {return this.md5}
}

// Main class
//exports.Label = function (texto, fuente, tam, color, cb) {
function Label (texto, fuente, tam, color, cb) {
    var myImage = new Image(texto,fuente,tam,color);
    function valid () {
        /*
        Limitations:
        texto.length < 200
        fuente in [1, fonttypes[].length] (subset of the whole fonts space)
        tam in fontsizes[] subset
        color in listaRGBcolors[] subset
        */
        if ((texto.length >= 200) || (fuente < 1) || (fuente > fonttypes.length)) {
            return false;
        } else {
            switch (tam) {
                case "16":
                case "24":
                case "32":
                case "40":
                case "48":
                    return (RGBcolors[color] !== undefined);
                    break;
                default:
                    return false;
            }
        }
```

```
    }

    function labelBuild_png () {
        // parameter check was performed before, so we don't need to do it again

        var fontname = fonttypes[parseInt(fuente, 10) - 1];
        imgk.convert(['-density', '72x72', '-background', 'transparent', '-fill',
                      RGBcolors[color], '-gravity',  'Center', '-pointsize', tam,
                      '-font', basedir + fontname, 'label:' + texto, 'png:'],
            function (err, stdout) {
              if (err) myImage.setImage( ERROR_IMG ); // handcrafted error handling
              else myImage.setImage( new Buffer(stdout, 'binary') );
              cb (err, myImage);
      })
    } // labelBuild_png
    myImage.setMD5( md5(fonttypes[1 + parseInt(fuente,10)] + tam + color + texto) );
    if (!valid()) {
      myImage.setImage( ERROR_IMG );
      cb (true, myImage);
    }
    else labelBuild_png();
} // Label

module.exports.Label=Label;
module.exports.Image=Image;
```

6.3 Dockerfile

```
FROM tsr/fedora-node-devel
ADD fonts /fonts
ADD public /public
ADD error.png /
ADD theImageFactory.js /
ADD miniWebServer.js /
CMD node miniWebServer
```

Utilizaremos este Dockerfile para generar una nueva imagen mediante la orden:

```
docker build –t part3-server .
```

7 ANEXO: RECURSOS NECESARIOS PARA ESTA PRÁCTICA

1. Infraestructura de virtuales que incluya LINUX, docker, docker-compose, ssh
2. Puertos abiertos:

- `firewall-cmd --zone=public --add-port=8000-8100/tcp --permanent`
- `firewall-cmd --reload`

   Al ejecutar `firewall-cmd --list-all`, en la sección public, debe aparecer una línea ports: 8000-8100/tcp

3. En los equipos, RPMs instalados mediante un gestor de paquetes: nodejs, zeromq-devel, ImageMagick, ImageMagick-devel
4. Para los programas NodeJS, bibliotecas instaladas mediante npm: md5, zmq, imagemagick

8 ANEXO: CONTENIDO DEL ARCHIVO TSR_LAB3_MATERIAL.ZIP

Este archivo, al descomprimirse, da lugara 3 carpetas en un primer nivel (fedora-node, misitio y wordpress) y un Dockerfile asociado a `misitio`. Las ilustraciones a continuación muestran más detalles.