

TSR - PRÁCTICA 3. DESPLIEGUE DE SERVICIOS. DOCKER

ÍNDICE

1. Introducción	3
2. Sesión 1. Primeros pasos con Docker. Despliegue de CBW	5
2.1. Construyendo la imagen base con Ubuntu, NodeJS y ØMQ.....	5
2.2. Despliegue de las imágenes individuales de client/broker/worker	5
2.3. Despliegue del sistema CBW de la práctica 2	6
3. Sesión 2. Almacenamiento persistente y acceso remoto	8
3.1. Anotando los diagnósticos	8
3.2. El componente logger y su efecto en el broker.....	8
3.3. Nuevas dependencias de los componentes	9
3.4. Acceso a almacenamiento persistente desde logger.....	9
3.5. Despliegue conjunto del nuevo servicio CBWL	9
3.6. Acceso externo	10
4. Sesión 3: Despliegue de un servicio prefabricado	13
4.1. Servicio web basado en imagen de WordPress.....	13
5. Anexos	23
5.1. Previo: construir la imagen tsr-zmq.....	23
5.2. Anexo 1: CBW (básico).....	23
5.3. Anexo 2: CBWL (con logger).....	24

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, presentándole un ejemplo de herramientas y aproximaciones que puede emplear para abordar tales retos.

Esta práctica requiere conocimientos asociados al tema 4 (**Despliegue**), y depende de la práctica 2, **ØMQ**, especialmente en lo relativo al sistema *client-broker-worker* (**CBW**) con socket ROUTER-ROUTER, variante tolerante a fallos. Debido al uso de Docker, la práctica solo puede ser completada en instalaciones tales como las virtuales de portal (DSIC) o la imagen de VirtualBox disponible desde el inicio de este curso.

Las actividades se basan en un material ([tsr_lab3_material.zip](#)) que, al descomprimirse, dará lugar a varias carpetas. Dentro de este documento se mencionan archivos contenidos en dichas carpetas. En cada sesión encontraremos apartados con instrucciones seguidos de otros con cuestiones sobre los resultados o nuevas propuestas de cambio.

La **primera sesión** se refiere al manejo básico de Docker, y afecta a la generación de las diferentes imágenes necesarias para el despliegue del sistema **CBW** anteriormente mencionado.

En la **segunda sesión** se desea añadir un par de componentes al sistema anterior. En primer lugar un componente *logger* que nos obligará a acceder al almacenamiento persistente del anfitrión, y en segundo lugar añadiremos un *cliente externo* que se ejecutará en otro equipo de portal (*negocia* con un@ compañero@) que debe conectar con el bróker que tú has desplegado.



Debido a los recortes de temario ordenados como adaptación a la catástrofe de la *Dana*, la **tercera sesión no se imparte** y se elimina del material evaluable para el actual curso 2024/2025. El material de este documento no ha sido recortado por su utilidad.

La **tercera sesión** propone el despliegue de sistemas preconfigurados, discutiendo los casos en los que esta posibilidad pueda ser recomendable, y completando una instalación junto con la verificación de su funcionamiento.

Para esta práctica se necesita:

1. Los conocimientos necesarios acerca de NodeJS y ZeroMQ que han sido objeto de estudio hasta la fecha. Esta práctica se encuentra específicamente ligada al tema 4, **Despliegue de servicios**, donde se hace hincapié en la tecnología de contenerización que representa Docker.
2. El funcionamiento de los servidores virtuales de portal, descrito en el documento sobre "*Laboratorios TSR*" disponible en PoliformaT.
3. Los materiales accesibles en PoliformaT ([tsr_lab3_material.zip](#)) en el directorio correspondiente a la tercera práctica.



Los archivos de [tsr_lab3_material.zip](#) son siempre más *fiables* que los fragmentos de código editados y añadidos a este documento.

4. Una **máquina virtual de portal**, que contiene ya una instalación de Docker y permite realizar los ejercicios planteados en esta práctica. En caso de que el servicio (demonio) docker no se encuentre en marcha, deberás ejecutar:

```
systemctl start docker
```

1. INTRODUCCIÓN

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

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, 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. Una forma de comprobar que dicha relación está bien construida será sometiendo el servicio a operaciones de **escalado**.
4. Una aplicación distribuida real contempla mayor variedad de agentes y situaciones que las que vemos aquí, y por ello introducimos **variaciones** que interactúan con recursos del anfitrión, tanto en almacenamiento como en comunicaciones.
5. Cuando no disponemos de un conocimiento y control de los componentes a integrar para construir un servicio, podemos optar por sistemas *llave en mano* que requieren relativamente poco esfuerzo para su puesta en marcha.

Una gran parte de los conceptos que se ponen en juego en esta práctica tienen su base en el escenario descrito en el primer ejemplo del apartado 6.5.2 de la guía del alumno del tema 4, 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 tema 4, **Docker** nos provee de herramientas para preparar de forma reproducible toda la pila software necesaria para la instanciación de un componente.
2. Para resolver adecuadamente el segundo problema se necesita especificar la configurabilidad de cada componente. 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 informaciones contenidas en él.
 - Es imprescindible que el código a desplegar sea configurable; en caso contrario no se podrá adaptar a los detalles procedentes de cada despliegue concreto.
3. Para abordar la tercera necesidad descrita procederemos incrementalmente a partir del esquema **CBW** de la práctica 2 ya mencionado.
 - Inicialmente desplegaremos todos los componentes de un servicio manualmente, usando las informaciones de configuración como parámetros de las órdenes docker.

- Posteriormente automatizaremos esta actividad mediante **docker compose**. Esto nos obligará a entender los fundamentos de este nuevo programa, y la especificación necesaria para construir el fichero **docker-compose.yml** con la interrelación entre los componentes de nuestra aplicación distribuida.
- Añadiremos **nuevos componentes** y situaciones, modificaremos el código necesario en los otros componentes y trazaremos nuevos planes de despliegue.



El escalado forma parte de la funcionalidad ofrecida por **docker compose**. La dificultad principal radica en la adecuación de los componentes de la aplicación distribuida para permitir y aprovechar dicho escalado.

4. Sobre el sistema **CBW** realizamos varias actividades:

- Despliegue y pruebas del sistema, comprobando tanto el funcionamiento normal como la reacción ante situaciones de fallo de trabajadores.
- Añadir un tercer componente, el **logger**, que será usado por el bróker, formará parte del despliegue y tendrá acceso al sistema de ficheros del anfitrión.
- Permitir el acceso al sistema a un componente externo no administrado por Docker, y que podría ejecutarse en cualquier otro ordenador (**cliente externo**), q con acceso a los sockets necesarios del bróker.

2. SESIÓN 1. PRIMEROS PASOS CON DOCKER. DESPLIEGUE DE CBW

2.1. CONSTRUYENDO LA IMAGEN BASE CON UBUNTU, NODEJS Y ØMQ

Necesitamos el punto de partida con el que estamos familiarizados, pero aplicado a nuestros contenedores. El ejemplo 1 del apartado 6.5.2 de la guía del alumno del tema 4 resume cómo generar `tsr-zmq`. No busques este dockerfile entre el material suministrado porque debes escribirlo tú. Te incluimos el Dockerfile y la orden indicados.



En ocasiones encontramos código en texto **PDF**, como éste, que pretendemos aprovechar mediante *copiar y pegar*. Desgraciadamente no todos los símbolos de un PDF son *normales*, y existe la posibilidad de que no sean admitidos por el programa al que van destinados (intérprete de NodeJS, Docker, ...). Para mencionar tres casos comunes, la tilde ('), el guión (-) y el tabulador no siempre son lo que parecen. Es probable que debas editar el texto pegado para solucionar esos problemas.

2.1.1. DOCKERFILE

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

2.1.2. ORDEN

```
docker build -t tsr-zmq .
```

Comprueba que la imagen existe tras ejecutar la orden.

2.2. DESPLIEGUE DE LAS IMÁGENES INDIVIDUALES DE CLIENT/BROKER/WORKER

Debes generar las imágenes de los tres componentes, pero necesita unas **adaptaciones** respecto al código ori-

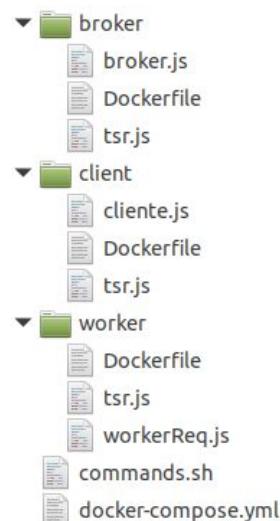
ginal de la práctica 2:

1. El identificador de clientes y trabajadores se calcula automáticamente desde su IP en lugar de obtenerse como argumento (esta adaptación ya se ha aplicado).
2. El código del cliente debe emitir 5 solicitudes antes de finalizar, lo que nos proporciona un volumen mayor de mensajes. Puedes inspirarte en el código del cliente externo, que emite 10 solicitudes.

Sin esta preparación los componentes no podrán interconectarse. En la ilustración de la derecha cada componente cuenta con su propia carpeta, con Dockerfile, código fuente y los archivos que necesite.

Los detalles se incluyen en el anexo correspondiente.

A continuación debes generar (*¿ya sabes cómo?*) 3 imágenes (imclient, imbroker, imworker), abrir 5 ventanas y ejecutar estas instrucciones para comprobar que todo encaja:



- Ventana 1: `docker run imbroker`
 - Mientras se encuentra en marcha, desde otra ventana, averigua y anota la IP que recibe el contenedor que ejecuta imbroker, y modifica adecuadamente los Dockerfiles de los otros.
 - La orden necesaria es `docker inspect`, pero debes averiguar el identificador del contenedor

Tras reajustar sus respectivos Dockerfiles:

- Ventanas 2 y 3: `docker run imworker`
- Ventanas 4 y 5: `docker run imclient`

2.3. DESPLIEGUE DEL SISTEMA CBW DE LA PRÁCTICA 2

En el apartado **6.7.4** de la guía del alumno del tema 4 se menciona cómo construir un despliegue orquestado de varios componentes para crear una aplicación (**CBW**) distribuida. La **tolerancia a fallos** se consigue mediante un temporizador que establece una ventana de tiempo en el que se espera la respuesta. Si no llega en ese intervalo, se interpreta que esa respuesta ya no llegará, y se reenvía este trabajo a otro trabajador.

Desde el material del apartado 1.2 de este boletín, únicamente necesitamos aplicar cambios en el archivo de despliegue (`docker-compose.yml`) para poder crear variables de entorno (`$BROKER_XXX`) que puedan aprovecharse en cada Dockerfile que deberás crear para clientes y trabajadores. El archivo resultante se encuentra en el anexo 1 (apartado 4.2.1).

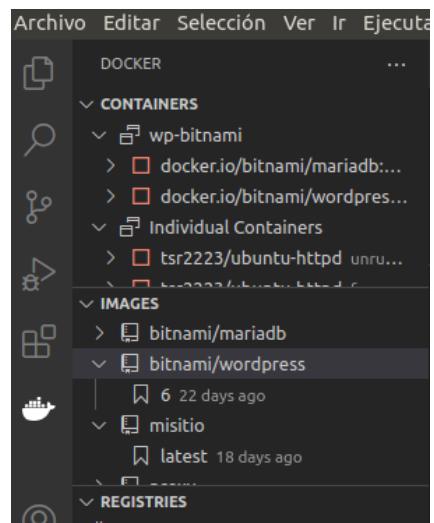
Para ejecutar 2 clientes y 5 trabajadores usaremos:

```
docker compose up --scale cli=2 --scale wor=5
```



Dado que los nombres de imágenes son globales, al avanzar en esta práctica será conveniente indicar a **docker compose** que no emplee las imágenes de los anteriores apartados. Usa para ello alguna o varias de las órdenes de **docker compose** (`down`, `kill`, `rm`, `rmi`)

Dispones del código y otras informaciones en el anexo correspondiente. Debes desplegarlo y escalar a 4 clientes y 2 trabajadores. Mientras se está ejecutando esta configuración, contesta las siguientes dos cuestiones:



Cuestión

Averigua la dirección IP de los 7 componentes desplegados (1 broker, 4 clientes y 2 trabajadores).

Cuestión

En la aplicación Visual Studio Code de las máquinas de portal se encuentra instalado el plugin para Docker. Observa toda la información que te puede proporcionar mientras se ejecutan los 7 componentes mencionados.

Comprueba que el funcionamiento es **correcto...**

- Cada cliente recibe respuesta a SU petición y no a las de otros
- No hay trabajadores libres si quedan peticiones pendientes
- Verificar (de nuevo) la tolerancia a fallos: hacer fallar un trabajador mientras atiende una petición y comprobar la reacción

Diseña modificaciones o formas de uso que faciliten las anteriores comprobaciones.

3. SESIÓN 2. ALMACENAMIENTO PERSISTENTE Y ACCESO REMOTO

3.1. ANOTANDO LOS DIAGNÓSTICOS

Los componentes deben mostrar diagnósticos para notificar cómo progresan y si hay incidencias. Es frecuente que una buena parte de esas notificaciones empleen la salida estándar, pero es una práctica extendida que, salvo urgencia, **esos diagnósticos se acumulen** cronológicamente en algún archivo para su posterior consulta; de hecho incluso existen formatos reconocidos para dichas anotaciones que permiten a aplicaciones externas *digerir* esa información. No es nuestro caso.

Podríamos elegir que cada componente guarde sus anotaciones en un fichero, pero no es cómodo contar con muchas fuentes de información. Si pretendiésemos que todos los componentes anotaran directamente sus diagnósticos en un único fichero centralizado, estaríamos completamente fuera de lugar... ¿un sistema distribuido con un fichero compartido?.

Como posible alternativa podemos desarrollar un componente (**logger**) capaz de recibir órdenes de escritura equivalentes a los `console.log()`

- Por simplicidad, solo usaremos este servicio desde el componente broker, pero es sencillo generalizarlo a todos los demás.

Aspectos destacables:

- Es importante que el archivo usado por el logger no pierda su contenido entre invocaciones. Recuerda que la naturaleza efímera de los contenedores es aquí un problema a resolver. Necesitarás usar un volumen Docker para conectar ese fichero con un espacio del anfitrión.
- Si algún componente debe considerar la existencia de este *logger*, deberá formar parte de su despliegue y será una dependencia a resolver.
- Un asunto interesante es el **tipo de socket ØMQ** aplicable: PULL para *logger* y PUSH para el resto será suficiente (aunque otras variaciones permitirían otras características).

3.2. EL COMPONENTE LOGGER Y SU EFECTO EN EL BROKER

Se comunica con el resto de procesos actuando como un recolector, con patrón PUSH-PULL. Su código, disponible en el anexo sobre CBWL, es sencillo para quien ya ha experimentado ØMQ. Además de la elección de sockets destaca la escritura en fichero.

Por claridad, **únicamente el broker** hace uso del servicio de anotaciones, por lo que el código y el despliegue del broker deberán tenerlo en cuenta. Necesitamos tres pequeñas cosas para construir este **broker** ...

- Incorporar en la línea de órdenes dos nuevos argumentos (`LoggerHost` y `loggerPort`)
- Un socket tipo PUSH para conectar con el componente logger

```
let slogger = zmq.socket('push')
conecta(slogger, loggerHost, loggerPort)
```

- Tras cada invocación a la función traza, añadir el envío del mismo texto (o algo similar) al logger

El Dockerfile del logger es prácticamente idéntico a otros ya estudiados, destacando el argumento con la ruta del directorio del contenedor (¡¡no lo confundas con el del anfitrión!!)

El código completo de ambos componentes se encuentra en el anexo 2.

3.3. NUEVAS DEPENDENCIAS DE LOS COMPONENTES

El broker necesitará conocer cómo conectar con *logger*. Esta situación es similar a la que ya relacionaba cliente y trabajador con el broker, y supone la necesidad de colocar una variable de entorno a sustituir en el despliegue. La última línea del Dockerfile del broker quedará:

```
CMD node mybroker 9998 9999 $LOGGER_HOST $LOGGER_PORT
```

3.4. ACCESO A ALMACENAMIENTO PERSISTENTE DESDE LOGGER

Es necesaria una consideración que no nos había preocupado hasta ahora: ¿cómo se relaciona el directorio con anotaciones (`/tmp/cbwlog`) del contenedor con el sistema de ficheros del anfitrión?. Mediante una sección `volumes`^[1] en la descripción del despliegue.

Supone que ya hemos creado el directorio `/tmp/logger.log` en el anfitrión

Si únicamente deseamos desplegar este componente, y no toda la aplicación distribuida, deberemos emplear una invocación de `docker run` con una opción equivalente a la sección `volumes`

```
docker run -v /tmp/logger.log:/tmp/cbwlog _parámetros_
```

3.5. DESPLIEGUE CONJUNTO DEL NUEVO SERVICIO CBWL

Al final del anexo 2 dispones de la parte significativa del nuevo `docker-compose.yml` que incluye el componente *logger*. También necesitarás un nuevo Dockerfile para **brokerl**.

La puesta en marcha con docker compose no tiene ninguna novedad. Es interesante que, desde el anfitrión, puede accederse a las anotaciones en el directorio `/tmp/logger.log`.

Prueba a realizar

El despliegue básico requiere un archivo de anotaciones vacío, y la ejecución de una combinación compuesta por 4 clientes, 2 trabajadores, 1 broker y 1 logger.

Cuestión

Detalla los pasos necesarios para cambiar la ubicación del fichero de log a un nuevo directorio y pruébalo.

Cuestión

¿Puedes modificar la función traza de `tsr.js`, empleada por el bróker, para incorporar el envío de mensajes al logger?

Cuestión

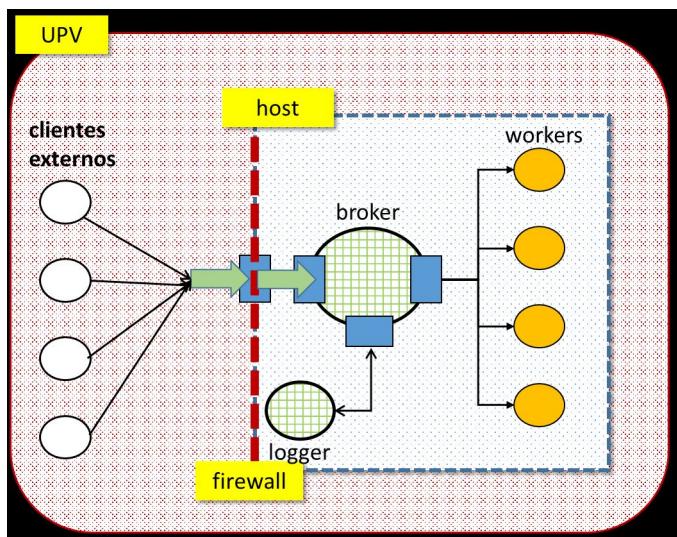
Reflexiona, sin necesidad de ejecutar, qué ocurriría si intentáramos desplegarlo en los siguientes escenarios:

- 2 clientes, 1 trabajador, 2 brokers, 1 logger
- 2 clientes, 1 trabajador, 1 broker, 2 loggers

Al final, ejecuta `docker compose down`

3.6. ACCESO EXTERNO

En una visión más realista, los clientes no son parte de la aplicación distribuida, y deberán interactuar con el servicio mediante algún punto bien conocido para poder encargarle trabajo. Un problema que aparece es la determinación del *endpoint* del servicio. Suponiendo que elijamos el URL del bróker:



- Si su IP puede cambiar en cada ejecución, no funcionará.
- Si los clientes se encuentran fuera del anfitrión que hospeda al resto de componentes, tendremos un problema de acceso (las IPs de Docker son locales dentro del anfitrión).

Ambos problemas pueden ser resueltos reservando un puerto del anfitrión que se hará corresponder con la IP y puertos del broker en el despliegue.

- Si el cortafuegos del anfitrión (*host*) se encuentra configurado correctamente, la línea `ports` de `docker-`

`compose.yml` hará el truco.

- Los equipos del portal en TSR suelen configurarse para permitir el acceso desde el exterior a los puertos 8000 a 9000. Para ampliar este soporte llegando al puerto 9999 ejecutamos:

```
ufw allow 8000:9999/tcp
```

Ahora podremos disponer de clientes externos que conectarán con el servicio mediante un URL fijo `tcp://hostIP:9998` que conducirá las peticiones al broker.

- Los **requisitos** de los clientes para su ejecución (NodeJS + ØMQ) **solo se satisfacen en las virtuales de portal y en las imágenes VirtualBox** proporcionadas.
 - Aunque algun@s alumn@s han tenido éxito añadiendo ØMQ a sus sesiones de **poliLabs**, esta experiencia no ha sido siempre positiva.
- Los clientes externos al anfitrión no pueden interactuar con el logger en esta configuración.

Convendrá realizar una prueba de funcionamiento con la versión de CBW que deseas, ejecutando el cliente externo en *otra máquina* de portal, pero considera estos pasos:

- Has adaptado la configuración de despliegue para añadir en la sección del broker:

```
ports:
- "9998:9998"
```

- En la *otra máquina* ejecutarás el/los cliente/s.
- También en esa *otra máquina* averiguarás la IP de tu servidor en el portal (p.ej. con una orden `ping tsr-miloggin-2425.dsicv.upv.es` que en este ejemplo suponemos que nos devuelve `172.23.105.111`)
 - Es importante mencionar que la IP del broker no es visible desde fuera del anfitrión, pero mediante la sección **ports** se ha ordenado al anfitrión que las peticiones entrantes por ese puerto se dirijan al mismo puerto del broker.

El código del cliente (en el directorio `cliente externo`) es idéntico^[2] al del cliente presentado en el sistema CBW, pero se ejecuta en la *otra máquina* de portal, lo que requiere que suministremos los argumentos necesarios.

- Tomando los datos del ejemplo, deberíamos invocarlo de esta forma:

```
node client_external 172.23.105.111 9998
```

- Y en el anfitrión arrancar el servicio mediante `docker compose up`.

Con estas pruebas y las modificaciones necesarias estamos empezando nuestra preparación para construir componentes que interactúan aunque no se encuentren en el mismo anfitrión.

Cuestión

Coloca algún texto significativo que luego puedas reconocer en la pantalla del bróker cuando procese un mensaje del cliente externo.

- [1] Dispones de información adicional en el material de referencia del tema 4
- [2] Por esta razón este apartado no posee entrada en los anexos

4. SESIÓN 3: DESPLIEGUE DE UN SERVICIO PREFABRICADO



Debido a los recortes de temario ordenados como adaptación a la catástrofe de la *dana*, la **tercera sesión no se imparte** y se elimina del material evaluable para el actual curso 2024/2025.

Pretendemos poner en marcha un servicio web, pero no somos especialistas en los componentes y tecnologías necesarios para construirlo, ni deseamos invertir mucho esfuerzo para conseguirlo. Es posible disponer de algún tutorial que nos indique, paso a paso, cómo construir y configurar cada pieza; sin embargo, esta alternativa no está libre de complicaciones: las instrucciones dependen de la fecha en que se redactan porque el software involucrado es muy variado y evoluciona con el tiempo. Esto provoca que los detalles, que en su momento encajaban, puedan no hacerlo en la actualidad.

Es un **problema de reproducibilidad** que ilustramos con dos ejemplos:

- Cuando en un Dockerfile se especifica una imagen base, como `FROM ubuntu:latest`, la versión de la imagen hace dos años no es la misma que la actual (ha pasado de la serie 22.X a la 24.X). La imagen generada **PUEDE NO SER LA MISMA**.
- Si la secuencia para generar la imagen incluye algo similar a `RUN apt-get upgrade`, la actualización que puede producir depende del software instalado y de las versiones más recientes. El resultado obtenido hace dos años no coincide con el que se obtendría ahora.

Esto nos debe llevar a fijar con precisión los componentes que intervienen y sus versiones, vigilando las posibles actualizaciones para que sigan funcionando como se espera. Esta responsabilidad no parece corresponder con nuestro papel "no especialista", lo que nos lleva a plantearnos como alternativa alguna solución *prefabricada* que discutimos a continuación.

4.1. SERVICIO WEB BASADO EN IMAGEN DE WORDPRESS

Para este apartado tomaremos un enfoque didáctico con el siguiente punto de partida: deseamos poner en marcha un servidor web basado en WordPress. Como no tenemos ninguna especificación en contra, implementaremos el servicio en un equipo con LINUX (gratuito), de manera que cumplimos a rajatabla el requisito LAMP de WordPress. Podemos averiguar cómo instalar y configurar este sistema como suma de sus piezas individuales, pero seguro que hay alternativas más sencillas y cómodas.

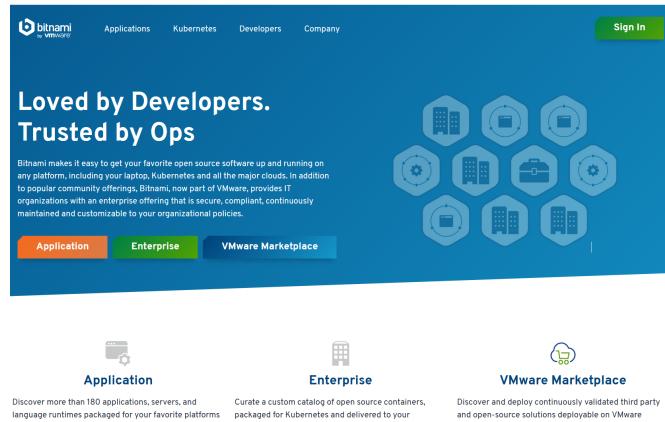
Nuestros primeros pasos suelen consistir en consultar en la web, empleando **wordpress** como término principal, destacando <https://es.wordpress.org/>. Además encontramos información sobre alojamiento, aplicaciones autoinstalables, libros, documentación, ... demasiadas^[1] referencias.

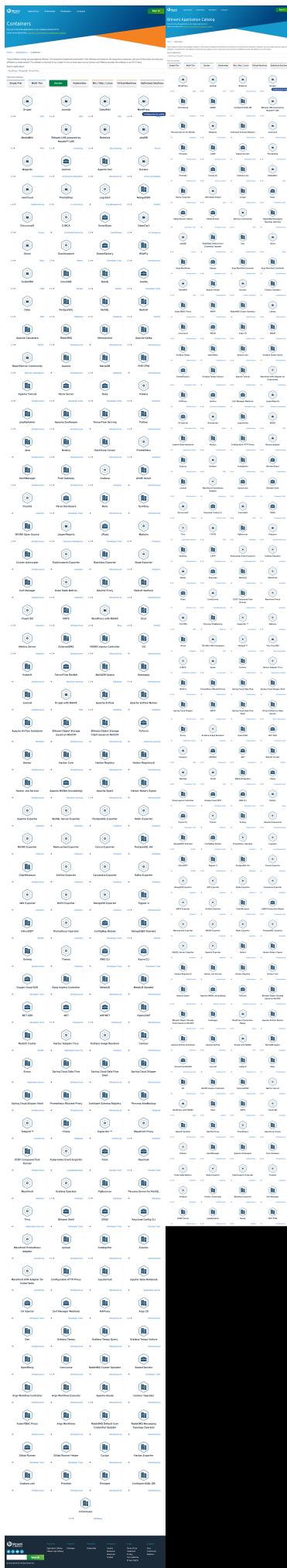
Ya que nos encontramos en el tema de despliegue de servicios, es oportuno aplicar las tecnologías que mencio-

namos a este caso, y enfocar el problema desde otra perspectiva: ¿un contenedor (o varios interconectados) para implementar un servicio con WordPress? Ahora el centro de las informaciones será Docker, y la búsqueda queda "docker wordpress" ya *solo* devuelve 62.200.000 de resultados (un 4.64% de la búsqueda anterior). La primera referencia que aparece es la de la imagen oficial (https://hub.docker.com/_/wordpress). En general toda esta web tiene un elevado interés si se desea localizar imágenes para contenedores, pero tomaremos otra alternativa menos conocida porque ilustra otras posibilidades: <https://bitnami.com/>

Bitnami es una empresa propiedad de VMware, una veterana especialista en virtualización. Dispone de un catálogo de aplicaciones/servicios, en su mayoría gratuitos, adaptados para varios tipos de virtualización, nubes, contenedores o ejecución en máquinas nativas.

El catálogo de aplicaciones consta de más de 180 entradas, y seleccionando la pestaña Docker también se observan centenares de elementos.





Buscando WordPress aparecen dos entradas: nos interesa la que **no** menciona nginx.

The screenshot shows the Bitnami Application Catalog interface. At the top, there's a navigation bar with links for Applications, Kubernetes, Developers, and Company, along with a Sign In button. Below the navigation is a section titled "Containers" with a sub-section "Find your favorite application in our catalog and launch it." It includes a note about trademarks and a search bar containing "wordpress". A "Search applications" button is also present. Below the search bar, there's a "Find your solution" section with categories: Single-Tier, Multi-Tier, Docker (which is highlighted in green), Kubernetes, Win / Mac / Linux, Virtual Machines, and Optimized Solutions. Two application cards are displayed: "WordPress" (4.5 stars) and "WordPress with NGINX" (4.0 stars). Both cards feature a hexagonal icon with a server-like symbol inside. Below each card is a "Blog" link. A "Configuración de cookies" button is located at the bottom right of the card area. At the very bottom of the page, there's a footer with links for Products (Application Catalog, VMware App Catalog), Projects (Kubeapps, Kubernetes), Solutions (Careers, Resources, Newsroom, Contact), Company (Terms of Use, Trademark, Privacy, Your California Privacy Rights), Legal (Docs, Community, Webinars), and Support. The footer also includes social media icons for Twitter, GitHub, and LinkedIn, and a "Search" bar.

[bitnami](#) by VMware Applications Kubernetes Developers Company Sign In

Home > Applications > Blog > WordPress packaged by Bitnami > Containers

WordPress packaged by Bitnami

4.5 (179 ratings) [www.wordpress.org](#)

Updated about 8 hours ago Version 6.1.0

DEPLOYMENT OFFERING

- On the cloud
- Containers
- On my computer
- Single-Tier
- Multi-Tier
- Docker**
- Kubernetes
- Win / Mac / Linux
- Virtual Machines

WORDPRESS PACKAGED BY BITNAMI CONTAINERS

Deploying Bitnami applications as containers is the best way to get the most from your infrastructure. Our application containers are designed to work well together, are extensively documented, and like our other application formats, our containers are continuously updated when new versions are made available.

Develop your applications in the same environment you will use on production
Up-to-date to the last version of the applications

Do you want to move your container to a Kubernetes infrastructure? Check out our [Helm charts](#).

CONTAINERS

 Azure	Available versions	GitHub Repository
Azure Container	6.0.3-1	Configuración de cookies

 Dockerhub Debian	Available versions	GitHub Repository
Dockerhub Debian	6.1.0-3	

INSTALLATION

Check the container documentation to find all the ways to run this application. We provide several docker-compose.yml configurations and other guides to run the image directly with docker.

[Getting started with WordPress packaged by Bitnami container](#)

PREREQUISITES

- Docker Engine 1.10.0
- Docker Compose is recommended with a version 1.6.0 or later

[bitnami](#) by VMware Products Application Catalog VMware App Catalog Projects Kubeapps Solutions Kubernetes Company Careers Resources Newsroom Contact Legal Terms of Use Trademark Privacy Your California Privacy Rights Support Docs Community Webinars © 2022 Bitnami | All Rights Reserved.

Las instrucciones para la instalación del contenedor se encuentran en <https://github.com/bitnami/containers/tree/main/bitnami/wordpress#how-to-use-this-image>, y se pueden resumir en...

```
curl -sSL
https://raw.githubusercontent.com/bitnami/containers/main/bitnami/wordpress/docker-
compose.yml > docker-compose.yml
docker compose up -d
```

El archivo `docker-compose.yml` (aparece como `docker-compose_wordpress.yml` en el material suministrado) es:

```
version: '2'
services:
  mariadb:
    image: docker.io/bitnami/mariadb:10.6
    volumes:
      - 'mariadb_data:/bitnami/mariadb'
    environment:
      # ALLOW_EMPTY_PASSWORD is recommended only for development.
      - ALLOW_EMPTY_PASSWORD=yes
      - MARIADB_USER=bn_wordpress
      - MARIADB_DATABASE=bitnami_wordpress
```

```

wordpress:
  image: docker.io/bitnami/wordpress:6
  ports:
    - '80:8080'
    - '443:8443'
  volumes:
    - 'wordpress_data:/bitnami/wordpress'
  depends_on:
    - mariadb
  environment:
    # ALLOW_EMPTY_PASSWORD is recommended only for development.
    - ALLOW_EMPTY_PASSWORD=yes
    - WORDPRESS_DATABASE_HOST=mariadb
    - WORDPRESS_DATABASE_PORT_NUMBER=3306
    - WORDPRESS_DATABASE_USER=bn_wordpress
    - WORDPRESS_DATABASE_NAME=bitnami_wordpress
volumes:
  mariadb_data:
    driver: local
  wordpress_data:
    driver: local

```

De manera que, con lo que sabemos de Docker, podemos predecir lo que sucederá la primera vez que ejecutemos la orden `docker compose up -d`, ¿no...?

```

[+] Running 5/5
  □ wordpress Pulled                                29.1s
  □ 75d78dd64cc0 Pull complete                      27.7s
  □ mariadb Pulled                                    21.0s
  □ 9b0425129f68 Pull complete                      6.3s
  □ a165d97ad6b6 Pull complete                      19.5s
[+] Running 5/5
  □ Network wp-bitnami_default Create...             0.2s
  □ Volume "wp-bitnami_mariadb_data" Created          0.0s
  □ Volume "wp-bitnami_wordpress_data" Created        0.0s
  □ Container wp-bitnami-mariadb-1 Started            1.4s
  □ Container wp-bitnami-wordpress-1 Started          2.4s

```

Puedes ver diagnósticos mucho más detallados desde Visual Studio Code con [View Logs](#) aplicado, p.ej., al contenedor `docker.io/bitnami/wordpress-1`

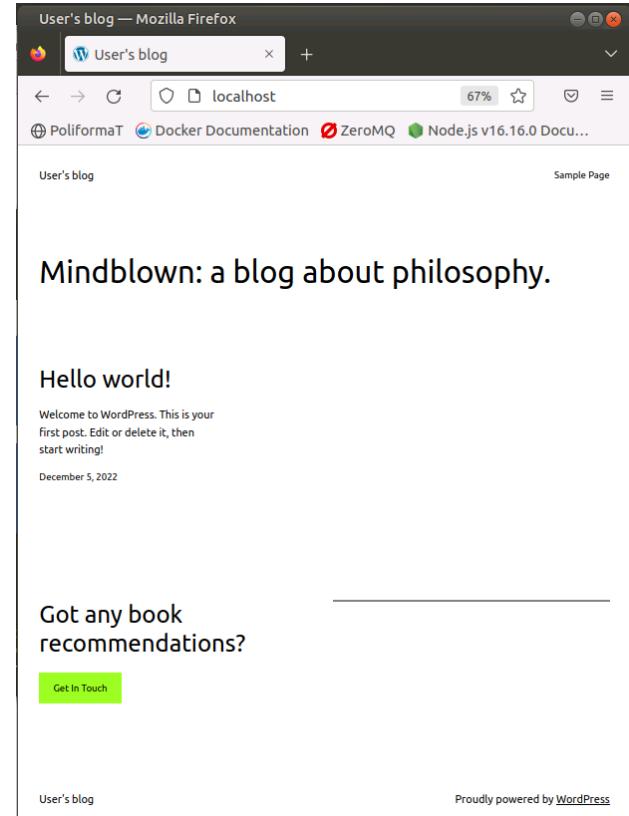
The screenshot shows the Docker Extension interface in a browser. On the left, there's a sidebar with sections like CONTAINERS, IMAGES, REGISTRIES, NETWORKS, VOLUMES, CONTEXTS, and HELP AND FEEDBACK. The main area is titled 'Inicio' and has tabs for PROBLEMAS, SALIDA, CONSOLA DE DEPURACIÓN, and TERMINAL. The TERMINAL tab is active, displaying the following log output:

```

* Ejecutando tarea: docker logs --tail 1000 -f 707edcda5630d77d4a0e286ce8e07772c2bb2ffd889d883ea249ef36ebd46a26
wordpress 23:07:22.49 Welcome to the Bitnami wordpress container
wordpress 23:07:22.49 Subscribe to project updates by watching https://github.com/bitnami/containers
wordpress 23:07:22.49 Submit issues and feature requests at https://github.com/bitnami/containers/issues
wordpress 23:07:22.49
wordpress 23:07:22.50 INFO ==> ** Starting WordPress setup **
realpath: /bitnami/apache/conf: No such file or directory
wordpress 23:07:22.54 INFO ==> Configuring Apache ServerTokens directive
wordpress 23:07:22.58 INFO ==> Configuring PHP options
wordpress 23:07:22.59 INFO ==> Setting PHP expose_php option
wordpress 23:07:22.63 INFO ==> Validating settings in MYSQL_CLIENT_* env vars
wordpress 23:07:22.71 WARN ==> You set the environment variable ALLOW_EMPTY_PASSWORD=yes. For safety reasons, do not use this flag in a production environment.
wordpress 23:07:23.42 INFO ==> Restoring persisted WordPress installation
wordpress 23:07:24.65 INFO ==> Trying to connect to the database server
wordpress 23:07:31.03 INFO ==> ** WordPress setup finished! **
wordpress 23:07:31.05 INFO ==> ** Starting Apache **
[Mon Dec 05 23:07:31.166127 2022] [ssl:warn] [pid 1] AH01909: www.example.com:8443:0 server certificate does NOT include an ID which match

```

Accedemos a este nuevo servidor desde el navegador web de nuestro anfitrión mediante la URL <http://localhost/>, o desde otro equipo con acceso al portal mediante <http://tsr-milogin-2425.dsicv.upv.es/>. La página resultante no tiene una gran apariencia, como se observa a la derecha.



Cuestión

He podido averiguar que, dentro del anfitrión, la URL <http://192.168.2.19:8080/> es equivalente a <http://localhost/>. ¿Cómo lo he podido averiguar? Reconstruye los pasos necesarios para hacerlo en tu virtual de portal.

WordPress permite acceder a la instalación en **modo edición**, pero exige que el usuario se identifique previamente. Usando la URI `/admin` y con las credenciales `user/bitnami` podemos pasar a administrar los contenidos del servidor mediante el *Dashboard*.

The screenshot shows the WordPress dashboard. At the top, there's a banner with the text "Welcome to WordPress!" and a link to "Learn more about the 6.1.1 version." Below the banner, there are three main sections: "Author rich content with blocks and patterns," "Customize your entire site with block themes," and "Switch up your site's look & feel with Styles." Each section has a brief description and a link to "Add a new page" or "Edit styles".

On the left side, there's a sidebar with navigation links: Home, Updates, Posts, Media, Pages, Comments, Appearance, Plugins, Users, Tools, and Settings. The "Pages" link is highlighted.

In the center, there are several widgets:

- Site Health Status:** Shows "No information yet..." and a note about site health checks.
- At a Glance:** Shows "1 Post", "1 Page", and "1 Comment". It also indicates "WordPress 6.1.1 running Twenty Twenty-Three theme."
- Activity:** Shows "Recently Published".
- Quick Draft:** A form for creating a new draft post, with fields for Title and Content, and a "Save Draft" button.
- WordPress Events and News:** A section for attending upcoming events, with a note that there are no events scheduled near the user at the moment.

En este punto realizamos una actividad sencilla: cambiar la página de ejemplo (enlazada con *Sample Page*). Para ello usamos la opción Pages en la columna izquierda, y pulsamos sobre Sample Page. A continuación modificamos su contenido.

The screenshot shows the "Pages" screen in the WordPress admin. The sidebar on the left has the "Pages" link selected. The main area displays a list of pages:

Title	Author	Date
Privacy Policy — Draft, Privacy Policy Page	user	Last Modified 2022/12/05 at 7:28 pm
Sample Page	user	Published 2022/12/05 at 7:28 pm

Below the list, there are "Bulk actions" and "Apply" buttons. The total count of items is 2.

This is an example page. It's different from a blog post because it will stay in one place and will show up in your site navigation (in most themes). Most people start with an [About page](#) that introduces them to potential site visitors. It might say something like this:

Esto lo he cambiado, pero tampoco me preocupa excesivamente...

Add citation

...or something like this:

The XYZ Doohickey Company was founded in 1971, and has been providing quality doohickeys to the public ever since. Located in Gotham City, XYZ employs over 2,000 people and does all kinds of awesome things for the Gotham community.

As a new WordPress user, you should go to [your dashboard](#) to delete this page and create new pages for your content. Have fun!

Pulsamos *Update* para que el cambio se aplique, y ya podemos acceder al contenido de la forma habitual (p.ej. con <http://localhost/>), pulsar en el enlace arriba a la derecha (ha cambiado su título) y verificar la nueva información.



Nuestro objetivo NO es el uso de WordPress ni la creación de contenidos. No inviertas mucho tiempo en rematar detalles de apariencia.

Una vez realizada esta acción, detenemos el servicio (`docker compose down`) y luego lo volvemos a lanzar (`docker compose up -d`). ¿Qué crees que va a pasar?

```
> docker compose down
[+] Running 3/3
  □ Container wp-bitnami-wordpress-1 Removed 0.8s
  □ Container wp-bitnami-mariadb-1 Removed 0.5s
  □ Network wp-bitnami_default Removed
```

La información y configuración que hemos modificado todavía *sigue ahí*.

Cuestión

Examina el archivo `docker-compose.yml` para determinar en qué lugar del anfitrión se guardan los contenidos de la base de datos. En ausencia de un lugar específico, examina `/var/lib/docker/volumes` (puede requerir privilegios mediante sudo).

Cuestión final

Puedes observar que el sistema desplegado es un sistema LAMP compuesto por dos piezas diferenciables. Recuerda que en el primer tema, a propósito de la Wikipedia, se mencionó la posibilidad de escalado separando el servidor APACHE de otro dedicado al SGBD. En el sistema WordPress contemplado en este apartado se puede aplicar la misma separación, colocando el servicio **wordpress** en un contenedor de un

equipo, y la BD **mariadb** en otro (deberías colaborar con un/a compañero/a), de forma que el contenedor *wordpress* actúe como cliente externo de *mariabd*. ¿Puedes *esbozar* los pasos necesarios para conseguirlo? No se pretende un despliegue simultáneo.

[1] Google, con fecha 20 de noviembre de 2024, informa sobre 1.340.000.000 resultados

5. ANEXOS

5.1. PREVIO: CONSTRUIR LA IMAGEN TSR-ZMQ

Esta imagen es el punto de partida para los servidores de portal. Debemos construirla, en caso de que no se haya hecho todavía, siguiendo los pasos del apartado 2.1.

5.2. ANEXO 1: CBW (BÁSICO)

Mantenemos sin cambios el código de los 3 componentes. Los Dockerfile de cliente y trabajador están parametrizados y así se facilita la resolución de dependencias.

5.2.1. DOCKER-COMPOSE.YML

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

Cuando los Dockerfile estén preparados, desplegar con docker compose up

5.2.2. CLIENTE Y SU DOCKERFILE

cliente.js

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

```

6 conecta(req, brokerHost, brokerPort)
7
8 req.send("C_" + require('os').hostname())
9
10 function procesaRespuesta(msg) {
11     traza('procesaRespuesta', 'msg', [msg])
12     adios([req], `Recibido: ${msg}. Adios`())
13 }
14 req.on('message', procesaRespuesta)
15 req.on('error', (msg) => {error(` ${msg}`)})
16 process.on('SIGINT', adios([req], "abortado con CTRL-C"))

```

En su Dockerfile únicamente cambiamos la última línea

```
CMD node myclient $BROKER_HOST $BROKER_PORT
```

Construir con `docker build`

5.2.3. WORKER Y SU DOCKERFILE

`worker.js`

```

1 const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
2 lineaOrdenes("brokerHost brokerPort")
3 let req = zmq.socket('req')
4 let id = "W_" + require('os').hostname()
5 req.identity = id
6
7 conecta(req, brokerHost, brokerPort)
8 req.send([' ', ' ', ' '])
9
10 function procesaPeticion(cliente, separador, mensaje) {
11     traza('procesaPeticion', 'cliente separador mensaje', [cliente, separador,
12     mensaje])
13     setTimeout(()=>{req.send([cliente, ' ', `${mensaje} ${id}`]), 1000})
14 }
15 req.on('message', procesaPeticion)
16 req.on('error', (msg) => {error(` ${msg}`)})
17 process.on('SIGINT', adios([req], "abortado con CTRL-C"))

```

En su Dockerfile únicamente cambiamos la última línea

```
CMD node myworker $BROKER_HOST $BROKER_PORT
```

Construir con `docker build`. Ya se puede desplegar y probar

5.3. ANEXO 2: CBWL (CON LOGGER)

En este anexo se proporciona información sobre el logger y el despliegue de CBW que se ajusta a este nuevo caso.

5.3.1. BROKERL Y SU DOCKERFILE

Solo necesitamos incorporar tres cosas al código de **brokerl**:

1. El nuevo par IP/puerto del logger, como últimos argumentos

```
lineaOrdenes("frontendPort backendPort loggerHost loggerPort")
let slogger = zmq.socket('push')
```

2. El socket de comunicación PUSH, y la conexión con el logger

```
conecta(slogger, loggerHost, loggerPort)
```

3. Y seleccionar los momentos en los que enviar la información al logger. En este ejemplo se aplica a la entrada a los listeners del bróker para sus dos routers

- En frontend: `slogger.send("frontend: cl="client", msg="+message)`
- En backend: `slogger.send("backend: wk="worker", cl="client", msg="+message)`

5.3.2. LOGGER Y SU DOCKERFILE

logger.js

```
1 const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require(''
  ./tsr')
2 lineaOrdenes("loggerPort filename")
3 // logger in NodeJS
4 // First argument is port number for incoming messages
5 // Second argument is file path for appending log entries
6
7 const fs = require('fs');
8 let log = zmq.socket('pull')
9
10 creaPuntoConexion(log, loggerPort)
11 log.on('message', (text) => {fs.appendFileSync(filename, text+'\n')})
```

En su Dockerfile deben cumplirse con los requisitos en anfitrión: directorio preexistente con permisos

```
FROM tsr-zmq
COPY ./tsr.js tsr.js
RUN mkdir logger
WORKDIR logger
COPY ./logger.js mylogger.js
VOLUME /tmp/cbwlog
EXPOSE 9995
CMD node mylogger 9995 $LOGGER_DIR/logs
```

5.3.3. DOCKER-COMPOSE.YML

Todo lo relacionado con clientes y trabajadores permanece igual, pero hemos de dar entrada al nuevo servicio logger, y a las nuevas dependencias para que...

- El broker pueda conectar con el logger (`$LOGGER_HOST` y `$LOGGER_PORT`)
- El logger pueda conocer el directorio de trabajo cedido por el host (`$LOGGER_DIR`)

A continuación se muestra el contenido de `docker-compose.yml`, del que interesa especialmente los servicios `bro` y `log`

```
version: '2'
services:
  cli:
    image: client
    build: ./client/
    links:
      - bro
    environment:
      - BROKER_HOST=bro
      - BROKER_PORT=9998
  wotr:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_HOST=bro
      - BROKER_PORT=9999
  bro:
    image: brokerl
    build: ./broker/
    expose:
      - "9998"
      - "9999"
    environment:
      - LOGGER_HOST=log
      - LOGGER_PORT=9995
  log:
    image: logger
    build: ./logger/
    expose:
      - "9995"
    volumes:
      # /tmp/logger.log DIRECTORY must exist on host and writeable
      - /tmp/logger.log:/tmp/cbwlog
    environment:
      - LOGGER_DIR=/tmp/cbwlog
```