TSR - PRÁCTICA 3
CURSO 2023/24

DESPLIEGUE DE SERVICIOS

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 algún otro equipo que deberá conectar con el anfitrión del bróker desplegado.

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. Conocer 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.

IMPORTANTE: Los archivos de `tsr_lab3_material.zip` son siempre más *fiables* que los fragmentos de código editados y añadidos a este documento.

4. Otra **máquina virtual de portal**, prestada momentáneamente por algún compañer@, para ejecutar el cliente externo de la segunda sesión.

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 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 que no se ejecuta en el anfitrión (**cliente externo**), que pueda acceder a los sockets necesarios del bróker.

1 Sesión 1. Primeros pasos con Docker. Despliegue de CBW

1.1 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`. Te incluimos el Dockerfile y la orden indicados.

1.1.1 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
```

1.1.2 Orden

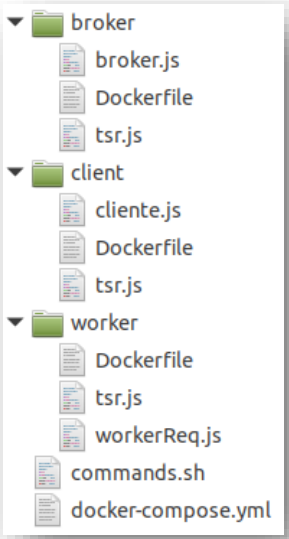
```
docker build -t tsr-zmq .
```

Comprueba que la imagen existe tras ejecutar la orden.

1.2 Despliegue de las imágenes individuales de client/broker/worker

Debes generar las imágenes de los tres componentes, pero necesita unas **adaptaciones QUE YA SE HAN REALIZADO** respecto al código original de la práctica 2:

1. El identificador de clientes y trabajadores se calcula automáticamente desde su IP en lugar de obtenerse como argumento.
2. El código del cliente debe emitir 5 solicitudes antes de finalizar, lo que nos proporciona un volumen mayor de mensajes.



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`

1.3 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.

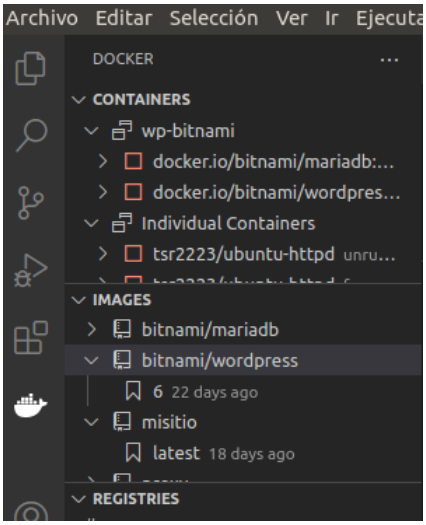
Desde el material del apartado 1.2 de este boletín, únicamente necesitamos aplicar cambios en el archivo de despliegue (`docker-compose.yml`) para poder crear variables de entorno ($BROKER_XXX) que puedan aprovecharse en cada Dockerfile que deberás crear para clientes y trabajadores. El archivo resultante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

2 SESIÓN 2. ALMACENAMIENTO PERSISTENTE Y ACCESO REMOTO

2.1 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).

2.2 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 **brokerl** …

- 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.

2.3 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
```

2.4 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
```

2.5 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

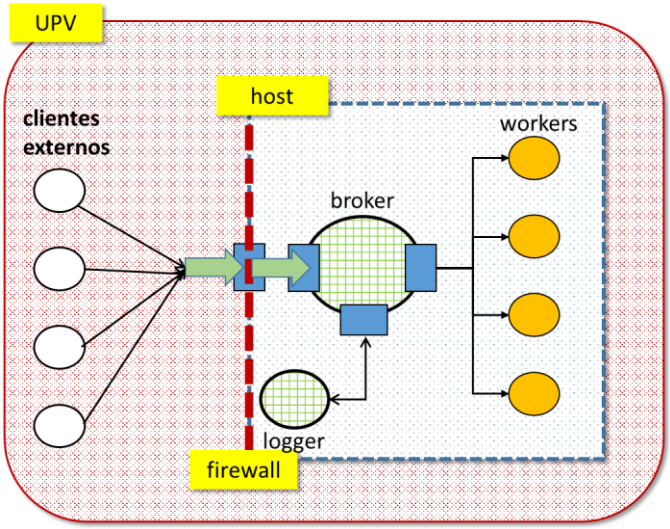
[1] Dispones de información adicional en el material de referencia del tema 4

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

Al final, ejecuta `docker-compose down`

2.6 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.
- Normalmente 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 desees, pero considera estos pasos:

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

```
ports:
```

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

2. En el otro equipo ejecutarás el/los cliente/s.
3. También en ese equipo averiguarás la IP de tu servidor en el portal (p.ej. con una orden `ping tsr-`*`milogin`*`-2324.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 otro equipo, lo que requiere que suministremos los argumentos necesarios.

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

```
node client_external 172.23.105.111 9998
```

5. 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.

3 Sesión 3: Despliegue de un servicio prefabricado

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 algunos años no es la misma que la actual, de manera que el resultado obtenido **PUEDE FALLAR** o **SER DIFERENTE**.
- 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 algunos 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

[2] Por esta razón este apartado no posee entrada en los anexos

responsabilidad no parece corresponder con nuestro papel "no especialista", lo que nos lleva a plantearnos como alternativa alguna solución *prefabricada* que discutimos a continuación.

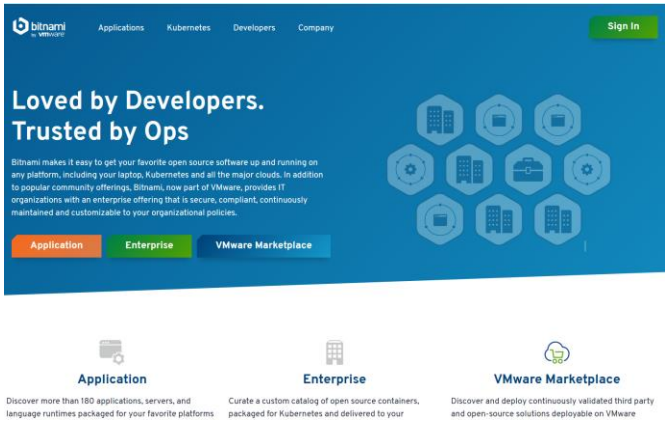
3.1 Servicio Web basado en imagen de WordPress

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[3] referencias.

Ya que nos encontramos en el tema de despliegue de servicios, es oportuno aplicar las tecnologías que mencionamos 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 "docker wordpress" ya *solo* devuelve 18.000.000 de resultados (un 1.76% 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.

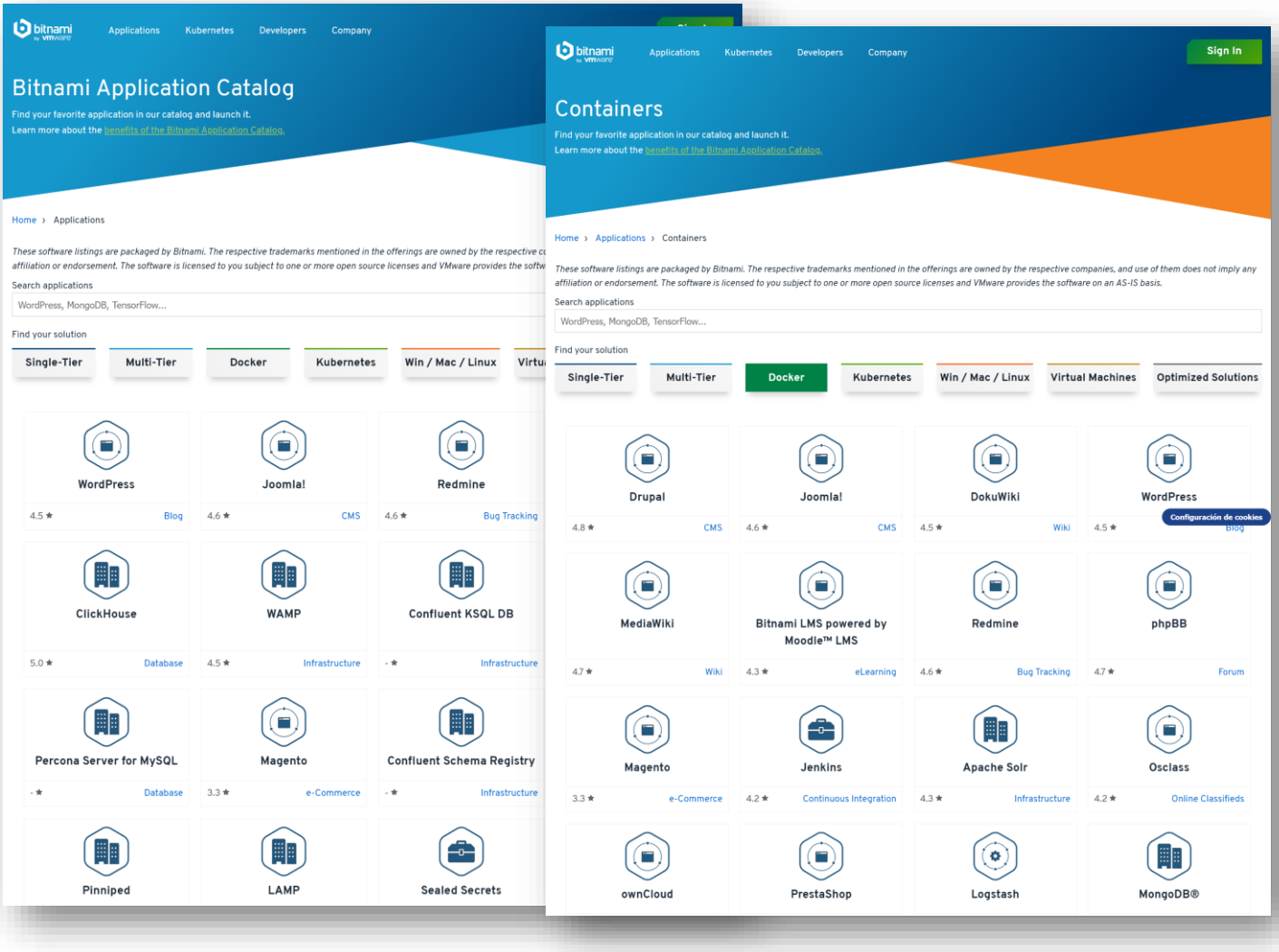


Dispone de un catálogo de aplicaciones/servicios, en su mayoría gratuitos, adaptados para varios tipos de virtualización, nubes, contenedores o ejecución en máquinas nativas..

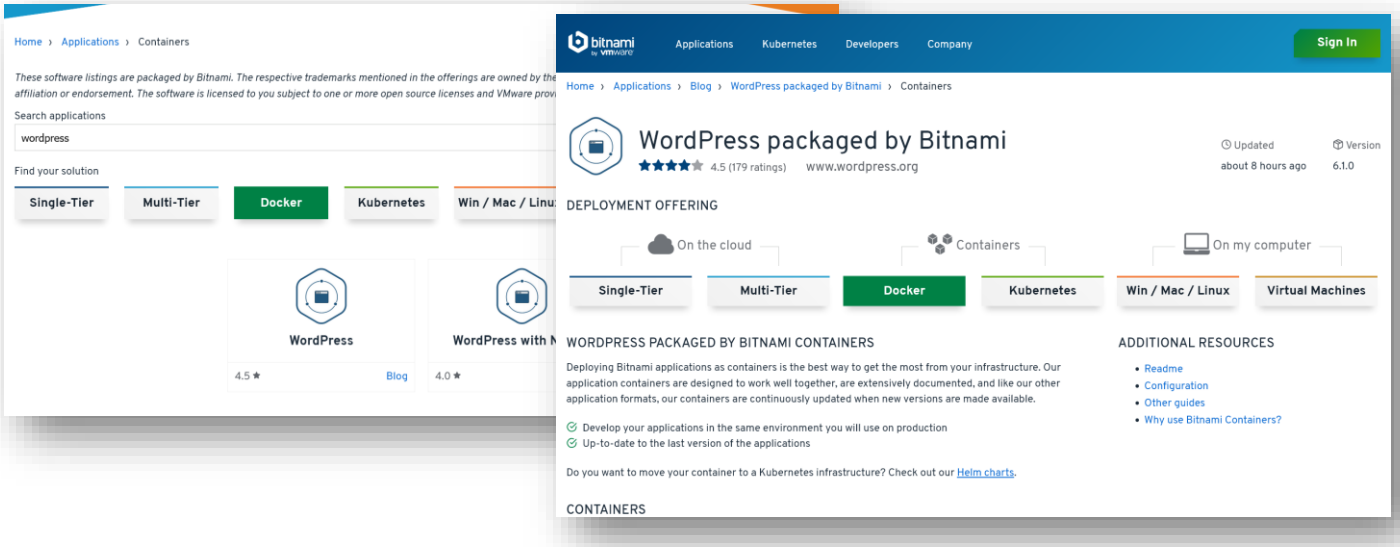
El catálogo de aplicaciones consta de más de 180 entradas, y seleccionando la pestaña Docker también se observan centenares de elementos.

- Encontrarás más información en el último anexo.

[3] Google, con fecha **9 de noviembre de 2022**, informa sobre 1.020.000.000 resultados



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



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
```

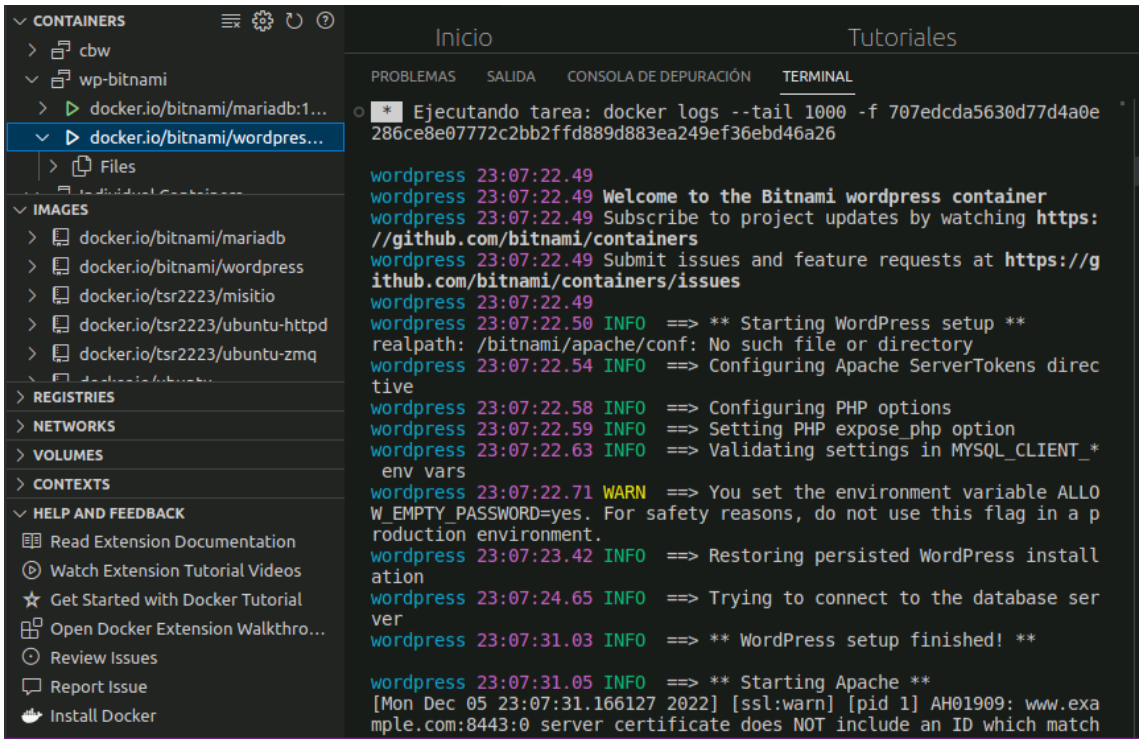
El archivo `docker-compose.yml` referenciado es:

```
version: '2'
services:
  mariadb:
    image: docker.io/bitnami/mariadb:11.1
    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
```

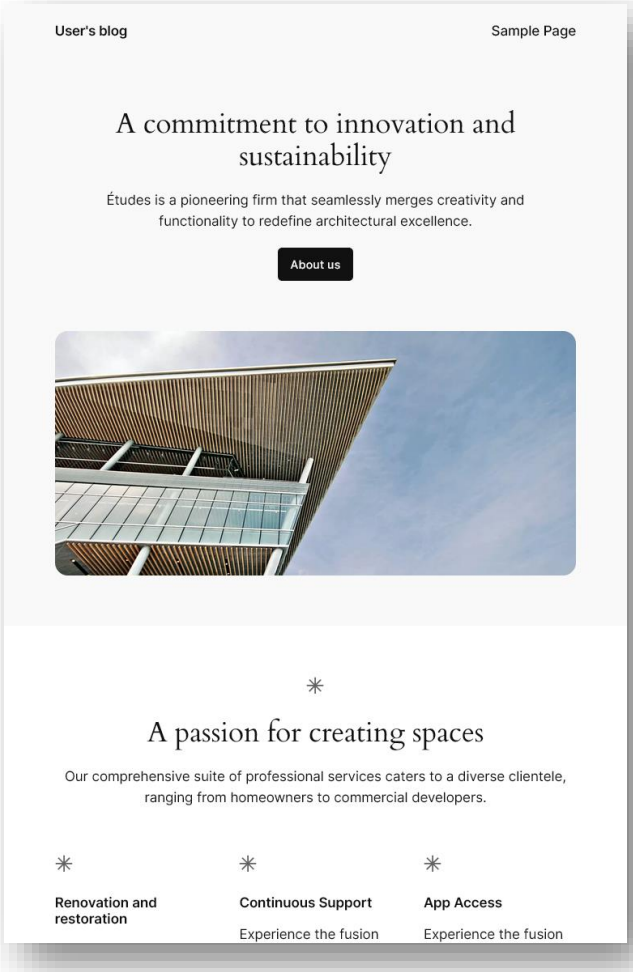
De manera que, con lo que sabemos de Docker, podemos predecir lo que sucederá la primera vez que ejecutemos la orden `docker-compose up -d`, ¿no…?

```
[+] 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          Cre...                 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`

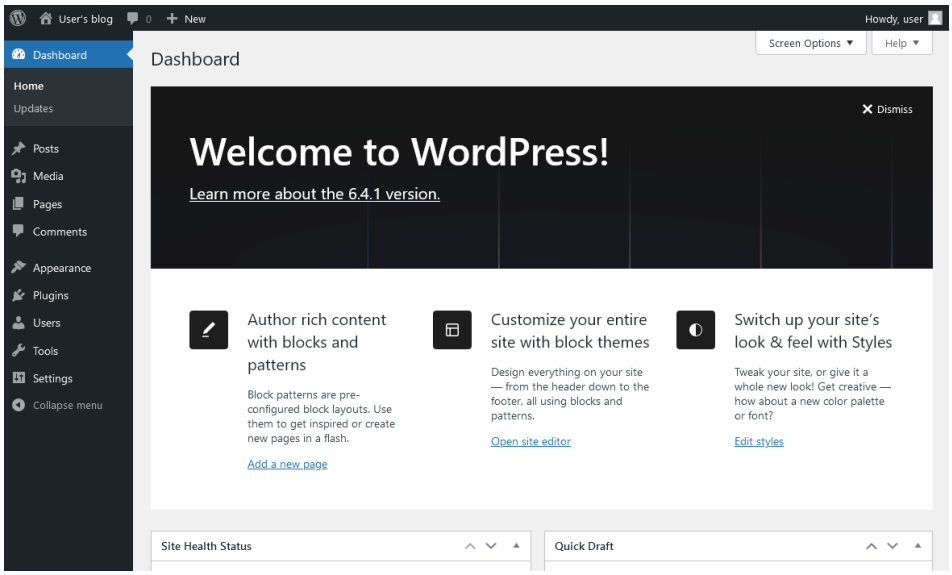


Accedemos a este nuevo servidor desde el navegador web de nuestro anfitrión mediante la URL `http://localhost/`, o desde otro equipo con acceso al portal mediante `http://tsr-`*`milogin`*`-2324.dsicv.upv.es/`. La página resultante se puede observar a la derecha.

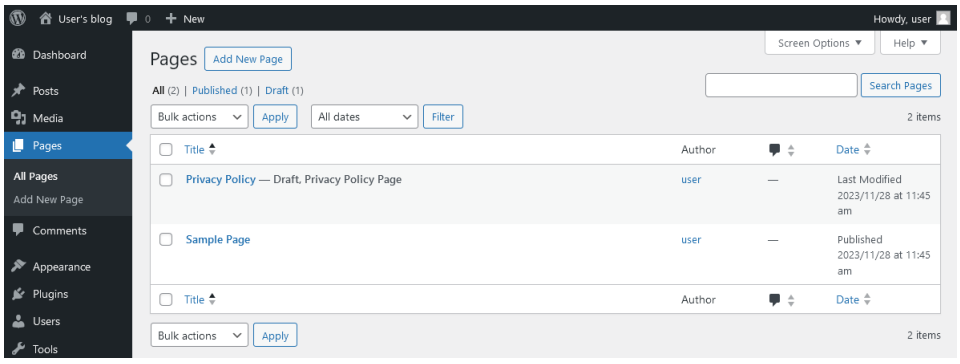


Cuestión: He podido averiguar que, dentro del anfitrión, la URL `http://192.168.2.19:8080/` es equivalente a `http://localhost/`. ¿Cómo lo he podido averiguar? Reconstruye los pasos necesarios para hacerlo en tu virtual de portal.

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*.



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[4] su contenido.



[4] No se muestran los detalles intermedios

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.

> **Nota**: 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.

4 ANEXOS

4.1 Anexo 0 (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.

4.1.1 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
```

4.1.2 Orden

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

4.2 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.

4.2.1 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`

4.2.2 Cliente y Dockerfile

cliente.js

```
01: const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
02: lineaOrdenes("brokerHost brokerPort")
03: let req = zmq.socket('req')
04: let id  = "C_"+require('os').hostname()
05: req.identity = id
06: conecta(req, brokerHost, brokerPort)
07:
08: req.send("C_"+require('os').hostname())
09:
10: var nMsgs = 5
11: function procesaRespuesta(msg) {
12:         traza('procesaRespuesta','msg',[msg])
13:         if (--nMsgs == 0) adios([req], `Recibido: ${msg}. Adios`)()
14: }
15: req.on('message', procesaRespuesta)
16: req.on('error', (msg) => {error(`${msg}`)})
17: process.on('SIGINT', adios([req],"abortado con CTRL-C"))
```

En su Dockerfile únicamente cambiamos la última línea

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

Construir con `docker build`

4.2.3 Worker y Dockerfile

worker.js

```
01: const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
02: lineaOrdenes("brokerHost brokerPort")
03: let req = zmq.socket('req')
04: let id  = "W_"+require('os').hostname()
05: req.identity = id
06:
07: conecta(req, brokerHost, brokerPort)
08: req.send(['','',''])
09:
10: function procesaPeticion(cliente, separador, mensaje) {
11:         traza('procesaPeticion','cliente separador mensaje',[cliente, separador, mensaje])
12:         setTimeout(()=>{req.send([cliente,'',`${mensaje} ${id}`])}, 1000)
13: }
14: req.on('message', procesaPeticion)
15: req.on('error', (msg) => {error(`${msg}`)})
16: process.on('SIGINT', adios([req],"abortado con CTRL-C"))
```

En su Dockerfile únicamente cambiamos la última línea

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

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

4.3 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.

4.3.1 Brokerl y 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)`

4.3.2 Logger y Dockerfile

`logger.js`

```
01: const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
02: lineaOrdenes("loggerPort filename")
03: // logger in NodeJS
04: // First argument is port number for incoming messages
05: // Second argument is file path for appending log entries
06:
07: const fs = require('fs');
08: let log = zmq.socket('pull')
09:
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 $LOGGER_DIR
EXPOSE 9995
CMD node mylogger 9995 $LOGGER_DIR/logs
```

4.3.3 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 de muestra el **fragmento significativo** de `docker-compose.yml`

```
version: '2'
services:

...

  bro:
    image: broker
    build: ./broker/
    links:
      - log
    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
```

4.4 Anexo 3: sobre el catálogo de imágenes en Bitnami

Con la proliferación del código libre es posible obtener implementaciones de servicios complejos con el mínimo coste económico. Solo se necesita conocimiento y esfuerzo para construir soluciones basadas en esos desarrollos.

Muchos productos procedentes del código libre han sido *envasados* en formas destinadas a su uso en máquinas físicas, máquinas virtuales, la nube o contenedores. En Internet encontramos depósitos, como el de Bitnami, en los que se pueden identificar dos tipos básicos de elementos:

- **Piezas** que pueden o deben ser combinadas con otras para implementar servicios o resolver problemas.
- **Sistemas** formados por múltiples piezas para dar solución, tras su configuración, a servicios o problemas concretos.

Para todas estas contribuciones podemos encontrar información procedente de varias fuentes: la web que aloja el desarrollo del proyecto (y que suele disponer de una comunidad de interesados), y las empresas u organizaciones que adaptan esos productos para su distribución.

A continuación mencionamos algunas piezas y sistemas destacables.

4.4.1 Piezas sueltas

Aquí podemos encontrar gestores de bases de datos relacionales y no relacionales, servidores y aceleradores web, distribuciones orientadas al desarrollo en determinados lenguajes, sistemas de almacenamiento en red, sistemas de mensajería, software de monitorización, servidores de autenticación, servidores de aplicaciones, etc.

Alguno de los productos con nombre propio son:

- Apache (servidor web, https://httpd.apache.org/)
- RabbitMQ (mensajería con broker, https://www.rabbitmq.com/)
- MariaDB (SGBD relacional, https://mariadb.org/)
- WildFly (servidor de aplicaciones Jakarta EE, https://www.wildfly.org/)
- Git (sistema de control de versiones, https://git-scm.com/)

4.4.2 Sistemas completos

Destacan muy especialmente los servicios basados en la web, que emplean HTTP como protocolo de acceso para sus clientes. Pueden citarse las publicaciones electrónicas, los gestores de contenido, las tiendas de comercio electrónico, los sistemas colaborativos[5], etc.

Alguno de los productos con nombre propio son:

- WordPress (CMS/blog, https://wordpress.com/es/)
- Alfresco (CMS, https://hub.alfresco.com/)
- Magento (comercio electrónico, https://developer.adobe.com/open/magento)
- Mastodon (redes sociales, https://joinmastodon.org/)

[5] Tanto de desarrollo de software como de edición colaborativa