

TSR 2425 - PRÀCTICA 3.  
DESPLEGAMENT DE SERVEIS.  
DOCKER

# ÍNDEX

1. Introducció .....	3
2. Sessió 1. Primers passos amb Docker. Desplegament de CBW .....	5
2.1. Construint la imatge base amb Ubuntu, NodeJS i ØMQ .....	5
2.2. Desplegament de les imatges individuals de client/broker/worker .....	5
2.3. Desplegament del sistema CBW de la pràctica 2 .....	6
3. Sessió 2. Emmagatzematge persistent i accés remot .....	8
3.1. Anotant els diagnòstics .....	8
3.2. El component logger i el seu efecte al broker .....	8
3.3. Noves dependències dels components .....	9
3.4. Accés a emmagatzematge persistent des de logger .....	9
3.5. Desplegament conjunt del nou servei CBWL .....	9
3.6. Accés extern .....	10
4. Sessió 3: Desplegament d'un servei prefabricat .....	13
4.1. Servei web basat en imatge de WordPress .....	13
5. Annexos .....	21
5.1. Previ: construir la imatge tsr-zmq .....	21
5.2. Annex 1: CBW (bàsic) .....	21
5.3. Annex 2: CBWL (amb logger) .....	22

El laboratori 3 es desenvoluparà al llarg de tres sessions. Els seus objectius principals són:



Que l'estudiant compregui alguns dels reptes que comporta el desplegament d'un servei multicomponent, presentant-li un exemple d'eines i aproximacions que pot fer servir per abordar aquests reptes.

Aquesta pràctica requereix coneixements associats al tema 4 (**Desplegament**), i depèn de la pràctica 2, **ØMQ**, especialment pel que fa al sistema *client-broker-worker* (**CBW**) amb socket ROUTER-ROUTER, variant tolerant a fallades. A causa de l'ús de Docker, la pràctica només es pot completar en instal·lacions com ara les virtuals de portal (DSIC) o la imatge de VirtualBox disponible des de l'inici d'aquest curs.

Les activitats es basen en un material ([tsr\\_lab3\\_material.zip](#)) que, en descomprimir-se, donarà lloc a diverses carpetes. Dins aquest document s'esmenten fitxers continguts en aquestes carpetes. A cada sessió trobarem apartats amb instruccions seguits dels altres amb qüestions sobre els resultats o noves propostes de canvi.

La **primera sessió** es refereix al maneig bàsic de Docker, i afecta la generació de les diferents imatges necessàries per al desplegament del sistema **CBW** anteriorment esmentat.

A la **segona sessió** es vol afegir un parell de components al sistema anterior. (*negocia* amb un@ company@) que ha de connectar amb el broker que tu has desplegat.



A causa de les retallades de temari ordenades com a adaptació a la catàstrofe de la *DANA*, la **tercera sessió no s'imparteix** i s'elimina del material avaluable per a l'actual curs 2024/2025. .

La **tercera sessió** proposa el desplegament de sistemes preconfigurats, discutint els casos en què aquesta possibilitat pugui ser recomanable, i completant una instal·lació juntament amb la verificació del seu funcionament.

Per a aquesta pràctica cal:

1. Els coneixements necessaris sobre NodeJS i ZeroMQ que han estat objecte d'estudi fins a la data. Aquesta pràctica es troba específicament lligada al tema 4, **Desplegament de serveis**, on es posa l'accent en la tecnologia de contenerització que representa Docker.
2. El funcionament dels servidors virtuals de portal, descrit al document sobre "*Laboratoris TSR*" disponible a PoliformaT.
3. Els materials accessibles a PoliformaT ([tsr\\_lab3\\_material.zip](#)) al directori corresponent a la tercera pràctica.



Els fitxers de [tsr\\_lab3\\_material.zip](#) són sempre més *fiables* que els fragments de codi editats i afegits a aquest document.

Una **màquina virtual de portal**, que conté ja una instal·lació de Docker i permet realitzar els exercicis plantejats en aquesta pràctica.

```
systemctl start docker
```

# 1. INTRODUCCIÓ

Els serveis són el resultat de l'execució d'una o diverses instàncies de cadascun dels components de programari emprats per implementar-los.

1. Un dels problemes en el moment del desplegament d'un servei és l'empaquetament de cadascun dels seus components de manera que la instanciació d'aquests components sigui repetible, i que l'execució de les instàncies de components s'aïlli de l'execució de la resta d'instàncies de qualsevol component.
2. Un altre problema a abordar consisteix en la **configuració** de cadascuna de les instàncies a desplegar.
3. També destaquem la necessitat d'especificar la **interrelació** entre els diferents components d'una aplicació distribuïda, especialment l'enllaç entre *endpoints*: com es poden definir i resoldre. Una manera de comprovar que aquesta relació està ben construïda serà sotmetent el servei a operacions d'escalat\*.
4. Una aplicació distribuïda real contempla més varietat d'agents i situacions que les que veiem aquí, i per això introduïm variacions que interactuen amb recursos de l'amfitrió, tant en emmagatzematge com en comunicacions.
5. Quan no disposem d'un coneixement i control dels components a integrar per construir un servei, podem optar per sistemes *clau en mà* que requereixen relativament poc esforç per posar-lo en marxa.

Una gran part dels conceptes que es posen en joc en aquesta pràctica tenen la base a l'escenari descrit al primer exemple de l'apartat 6.5.2 de la guia de l'alumne del tema 4, encara que també hi intervenen altres condicionants pràctics que no poden ser ignorats .

En aquest laboratori explorem maneres de construir components, configurar-los, connectar-los i executar-los per formar aplicacions distribuïdes escalables d'una manera *raisonablement* senzilla. Per això ens dotem de tecnologies especialitzades en aquest àmbit.

1. Ens enfrontem al primer problema amb l'ajuda del `_framework_Docker*`. Tal com s'ha estudiat en el tema 4, Docker ens proveeix d'eines per preparar de forma reproducible tota la pila programari necessària per a la instanciació d'un component.
2. Per resoldre adequadament el segon problema cal especificar la configurabilitat de cada component. Donada la nostra elecció de tecnologia (**Docker**) haurem d'entendre com donar a conèixer les dependències perquè el *framework* de Docker les resolgui. Concretament necessitarem conèixer com emplenar un **Dockerfile** i com referenciar informacions contingudes en ell.
  - És imprescindible que el codi a desplegar sigui configurable; en cas contrari, no es podrà adaptar als detalls procedents de cada desplegament concret.
3. Per abordar la tercera necessitat descrita procedirem incrementalment a partir de l'esquema CBW de la pràctica 2 ja esmentat.
  - Inicialment desplegarem tots els components d'un servei manualment, usant les informacions de configuració com a paràmetres de les ordres docker.
  - Posteriorment automatitzarem aquesta activitat mitjançant **docker compose**. Això ens obligarà a entendre els fonaments d'aquest nou programa i l'especificació necessària per construir el fitxer

**docker-compose.yml** amb la interrelació entre els components de la nostra aplicació distribuïda.

- Afegirem **nous components** i situacions, modificarem el codi necessari en els altres components i traçarem nous plans de desplegament.



L'escalat forma part de la funcionalitat oferta per **docker compose**. La dificultat principal rau en l'adequació dels components de l'aplicació distribuïda per permetre i aprofitar aquest escalat.

#### 4. Sobre el sistema **CBW** realitzem diverses activitats:

- Desplegament i proves del sistema, comprovant tant el funcionament normal com la reacció davant de situacions de fallada de treballadors.
- Afegir un tercer component, el **logger**, que serà usat pel broker, formarà part del desplegament i tindrà accés al sistema de fitxers de l'amfitrió.
- Permetre l'accés al sistema a un component extern no administrat per Docker, i que podria executar-se en qualsevol altre ordinador (**client extern**), q amb accés als sockets necessaris del broker.

## 2. SESSIÓ 1. PRIMERS PASSOS AMB DOCKER. DESPLEGAMENT DE CBW

### 2.1. CONSTRUINT LA IMATGE BASE AMB UBUNTU, NODEJS I ØMQ

Necessitem el punt de partida amb què estem familiaritzats, però aplicat als nostres contenidors. L'exemple 1 de l'apartat 6.5.2 de la guia de l'alumne del tema 4 resumeix com generar `tsr-zmq`. No busques aquest `dockerfile` entre el material subministrat perquè l'has d'escriure tu. Us incloem el `Dockerfile` i l'ordre indicats.



De vegades trobem codi en text **PDF**, com aquest, que pretenem aprofitar mitjançant *copiar i enganxar*. Malauradament no tots els símbols d'un PDF són *normals*, i hi ha la possibilitat que no siguin admesos pel programa al qual van destinats (intèrpret de NodeJS, Docker,...). Per esmentar tres casos comuns, la titlla (```), el guió (`-`) i el tabulador no sempre són el que semblen. És probable que hàgiu d'editar el text enganxat per solucionar aquests problemes.

#### 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 -i
RUN npm init -y
RUN npm install zeromq@5
```

#### 2.1.2. ORDRE

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

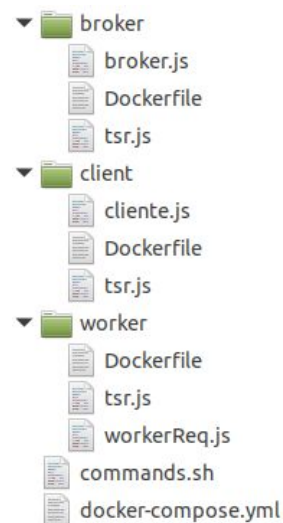
Comproveu que la imatge existeix després d'executar l'ordre.

## 2.2. DESPLEGAMENT DE LES IMATGES INDIVIDUALS DE CLIENT/BROKER/WORKER

Has de generar les imatges dels tres components, però necessita unes adaptacions respecte al codi original de la pràctica 2:

1. L'identificador de clients i treballadors es calcula automàticament des del vostre IP en lloc d'obtenir-se com a argument (aquesta adaptació ja s'ha aplicat).
2. El codi del client ha d'emetre 5 sol·licituds abans de finalitzar, cosa que ens proporciona un volum més gran de missatges. Pots inspirar-te en el codi del client extern, que emet 10 sol·licituds.

Sense aquesta preparació els components no es poden interconnectar. A la il·lustració de la dreta cada component compta amb la seva pròpia carpeta, amb Dockerfile, codi font i els fitxers que necessiti.



Els detalls s'inclouen a l'annex corresponent.

A continuació has de generar (*ja saps com?*) 3 imatges (imclient, imbroker, imworker), obrir 5 finestres i executar aquestes instruccions per comprovar que tot encaixa:

- Finestra 1: `docker run imbroker`
  - Mentre es troba en marxa, des d'una altra finestra, esbrina i anota la IP que rep el contenidor que executa imbroker, i modifica adequadament els Dockerfiles dels altres.
  - L'ordre necessària és `docker inspect`, però has d'esbrinar l'identificador del contenidor

Després de reajustar els respectius Dockerfiles:

- Finestres 2 i 3: `docker run imworker`
- Finestres 4 i 5: `docker run imclient`

## 2.3. DESPLEGAMENT DEL SISTEMA CBW DE LA PRÀCTICA 2

A l'apartat **6.7.4** de la guia de l'alumne del tema 4 s'esmenta com construir un desplegament orquestrat de diversos components per crear una aplicació (**CBW**) distribuïda. La **tolerància a errors** s'aconsegueix mitjançant un temporitzador que estableix una finestra de temps en què s'espera la resposta. Si no arriba en aquest interval, s'interpreta que aquesta resposta ja no arribarà, i es reenvia aquest treball a un altre treballador.

Des del material de l'apartat 1.2 d'aquest butlletí, només necessitem aplicar canvis al fitxer de desplegament (`docker-compose.yml`) per poder crear variables d'entorn (`$BROKER_XXX`) que puguin aprofitar-se a cada Dockerfile que hauràs de crear per clients i treballadors. L'arxiu resultant és a l'annex 1 (apartat 4.2.1).

Per executar 2 clients i 5 treballadors farem servir:

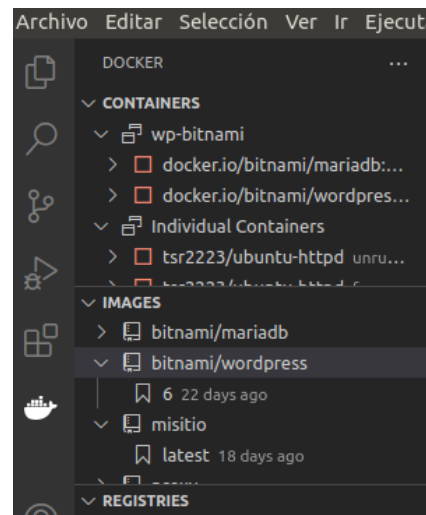
```
docker compose up --scale cli=2 --scale wor=5
```





Atès que els noms d'imatges són globals, en avançar en aquesta pràctica serà convenient indicar a **docker compose** que no faci servir les imatges dels anteriors apartats. Usa per a això alguna o diverses de les ordres de **docker compose** (**down**, **kill**, **rm**, **rmi**)

Disposes del codi i altres informacions a l'annex corresponent. Has de desplegar-ho i escalar a 4 clients i 2 treballadors. Mentre s'està executant aquesta configuració, contesta les dues qüestions següents:



### Qüestió

Esbrina l'adreça IP dels 7 components desplegats (1 broker, 4 clients i 2 treballadors).

### Qüestió

A l'aplicació Visual Studio Code de les màquines de portal hi ha instal·lat el plugin per a Docker. Observa tota la informació que us pot proporcionar mentre s'executen els 7 components esmentats.

Comprova que el funcionament és **correcte**...

- Cada client rep resposta a la SEVA petició i no a les dels altres
- No hi ha treballadors lliures si queden peticions pendents
- Verificar (de nou) la tolerància a fallades: fer fallar un treballador mentre atén una petició i comprovar la reacció

Dissenya modificacions o formes d'ús que faciliten les comprovacions anteriors.

## 3. SESSIÓ 2. EMMAGATZEMATGE PERSISTENT I ACCÉS REMOT

### 3.1. ANOTANT ELS DIAGNÒSTICS

Els components han de mostrar diagnòstics per notificar com progressen i si hi ha incidències. És freqüent que una bona part d'aquestes notificacions facin servir la sortida estàndard, però és una pràctica estesa que, llevat d'urgència, aquests diagnòstics s'acumulin cronològicament en algun arxiu per a la posterior consulta; de fet fins i tot hi ha formats reconeguts per a aquestes anotacions que permeten a aplicacions externes *digerir* aquesta informació. No és el nostre cas.

Podríem triar que cada component guardi les seves anotacions en un fitxer, però no és còmode comptar amb moltes fonts d'informació. un sistema distribuït amb un fitxer compartit?.

Com a possible alternativa podem desenvolupar un component (**logger**) capaç de rebre ordres d'escriptura equivalents als **console.log()**

- Per simplicitat, només farem servir aquest servei des del component broker, però és senzill generalitzar-lo a tots els altres.

Aspectes destacables:

- És important que el fitxer usat pel logger no perdi el contingut entre invocacions. Recordeu que la naturalesa efímera dels contenidors és aquí un problema a resoldre. Necessitareu utilitzar un volum Docker per connectar aquest fitxer amb un espai de l'amfitrió.
- Si algun component ha de considerar l'existència d'aquest *logger*, haurà de formar part del seu desplegament i serà una dependència a resoldre.
- Un assumpte interessant és el **tipus de socket ØMQ** aplicable: PULL per a *logger* i PUSH per a la resta serà suficient (encara que altres variacions permetrien altres característiques).

### 3.2. EL COMPONENT LOGGER I EL SEU EFECTE AL BROKER

Es comunica amb la resta de processos actuant com un recol·lector, amb patró PUSH-PULL. El codi, disponible a l'annex sobre CBWL, és senzill per a qui ja ha experimentat ØMQ. A més de l'elecció de sockets, destaca l'escriptura en fitxer.

Per claredat, **únicament el broker** fa ús del servei d'anotacions, per la qual cosa el codi i el desplegament del broker ho han de tenir en compte. Necessitem tres petites coses per construir aquest brokerl ...

- Incorporar a la línia d'ordres dos nous arguments (**loggerHost** i **loggerPort**)
- Un socket tipus PUSH per connectar amb el component logger

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

- Després de cada invocació a la funció `traça`, afegir l'enviament del mateix text (o alguna cosa similar) al `logger`

El Dockerfile del `logger` és pràcticament idèntic a altres ja estudiats, destacant l'argument amb la ruta del directori del contenidor (no ho confonguis amb el de l'amfitrió!!)

El codi complet dels dos components es troba a l'annex 2.

### 3.3. NOVES DEPENDÈNCIES DELS COMPONENTS

El broker necessitarà conèixer com connectar amb `logger`. Aquesta situació és similar a la que ja relacionava client i treballador amb el broker, i suposa la necessitat de col·locar una variable d'entorn a substituir al desplegament. L'última línia del Dockerfile del broker quedarà:

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

### 3.4. ACCÉS A EMMAGATZEMATGE PERSISTENT DES DE LOGGER

Cal una consideració que no ens havia preocupat fins ara: com es relaciona el directori amb anotacions (`/tmp/cbwlog`) del contenidor amb el sistema de fitxers de l'amfitrió? Mitjançant una secció `volumes`<sup>[1]</sup> a la descripció del desplegament.

**Suposa que ja hem creat el directori `/tmp/logger.log` a l'amfitrió**

Si només desitgem desplegar aquest component, i no tota l'aplicació distribuïda, haurem de fer servir una invocació de `docker run` amb una opció equivalent a la secció `volumes`

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

### 3.5. DESPLEGAMENT CONJUNT DEL NOU SERVEI CBWL

Al final de l'annex 2 disposeu de la part significativa del nou `docker-compose.yml` que inclou el component `logger`. També necessitaràs un nou Dockerfile per a `brokerl`.

La posada en marxa amb `docker compose` no té cap novetat. És interessant que, des de l'amfitrió, podeu accedir a les anotacions al directori `/tmp/logger.log`.

## Prova a realitzar

El desplegament bàsic requereix un fitxer d'anotacions buit, i l'execució d'una combinació composta per 4 clients, 2 treballadors, 1 broker i 1 logger.

## Qüestió

Detalla els passos necessaris per canviar la ubicació del fitxer de log a un directori nou i prova-ho.

## Qüestió

Pots modificar la funció `traça` de `tsr.js`, emprada pel broker, per incorporar l'enviament de missatges al logger?

## Qüestió

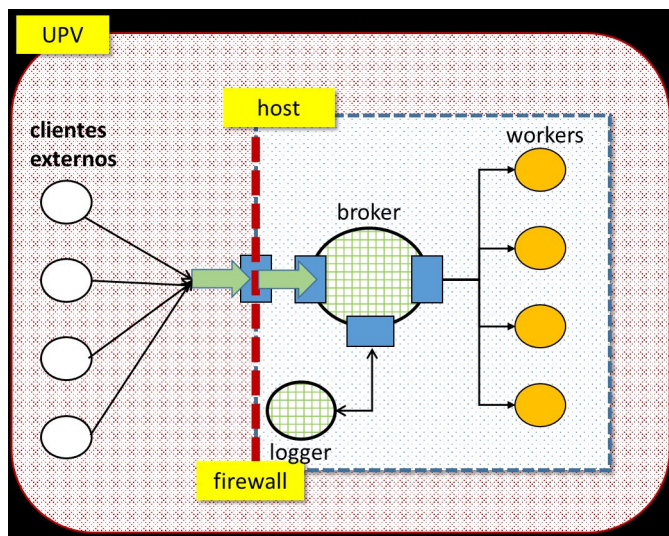
Reflexiona, sense necessitat d'executar, què passaria si intentéssim desplegar-ho als escenaris següents:

- 2 clients, 1 treballador, 2 brokers, 1 logger
- 2 clients, 1 treballador, 1 broker, 2 loggers

Al final, executa `docker compose down`

## 3.6. ACCÉS EXTERN

En una visió més realista, els clients no són part de l'aplicació distribuïda, i hauran d'interactuar amb el servei mitjançant algun punt ben conegut per poder encarregar feina. Un problema que apareix és la determinació del *endpoint* del servei. Suposant que triem l'URL del broker:



- Si la vostra IP pot canviar en cada execució, no funcionarà.
- Si els clients estan fora de l'amfitrió que allotja la resta de components, tindrem un problema d'accés (les IPs de Docker són locals dins de l'amfitrió).

Ambdós problemes poden ser resolts reservant un port de l'amfitrió que es farà correspondre amb la IP i ports del broker al desplegament.

- Si el tallafoc de l'amfitrió (*host*) està configurat correctament, la línia `ports` de `docker-compose.yml` farà el truc.
- Els equips del portal a TSR solen configurar-se per permetre l'accés des de l'exterior als ports 8000 a 9000.

Per ampliar aquest suport arribant al port 9999 executem:

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

Ara podrem disposar de clients externs que connectaran amb el servei mitjançant un URL fix **tcp://hostIP:9998** que conduirà les peticions al broker.

- Els **requisits** dels clients per a la seva execució (NodeJS + ØMQ) **només se satisfan a les virtuals de portal i a les imatges VirtualBox** proporcionades.
  - Encara que alguns alumnes han tingut èxit afegint ØMQ a les seves sessions de poliLabs, aquesta experiència no ha estat sempre positiva.
- Els clients externs a l'amfitrió no poden interactuar amb el logger en aquesta configuració.

Convindrà realitzar una prova de funcionament amb la versió de CBW que desitgis, executant el client extern a *una altra màquina* de portal, però considera aquests passos:

1. Heu adaptat la configuració de desplegament per afegir a la secció del broker:

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

2. A l'*una altra màquina* executaràs el/els client/s.
3. També en aquesta *una altra màquina* esbrinaràs la IP del teu servidor al portal (p.ex. amb una ordre **ping tsr-mi login-2425.dsicv.upv.es** que en aquest exemple suposem que ens torna **172.23.105.111**)
  - És important esmentar que la IP del broker no és visible des de fora de l'amfitrió, però mitjançant la secció ports s'ha ordenat a l'amfitrió que les peticions entrants per aquest port s'adreixin al mateix port del broker.

El codi del client (al directori **client extern**) és idèntic <sup>[2]</sup> al del client presentat al sistema CBW, però s'executa a la *una altra màquina* de portal, cosa que requereix que subministrem els arguments necessaris.

4. Prenent les dades de l'exemple, ho hauríem d'invocar d'aquesta manera:

```
node client_external 172.23.105.111 9998
```

5. I a l'amfitrió arrencar el servei mitjançant **docker compose up**.

Amb aquestes proves i les modificacions necessàries estem començant la nostra preparació per construir components que interactuen encara que no es trobin al mateix amfitrió.

## Qüestió

Col·loca algun text significatiu que després puguis reconèixer a la pantalla del broker quan processis un missatge del client extern.

[1] Disposeu d'informació addicional al material de referència del tema 4

[2] Per aquesta raó aquest apartat no posseeix entrada als annexos

## 4. SESSIÓ 3: DESPLEGAMENT D'UN SERVEI PREFABRICAT



A causa de les retallades de temari ordenades com a adaptació a la catàstrofe de la *DANA*, la **tercera sessió no s'imparteix** i s'elimina del material avaluable per a l'actual curs 2024/2025.

Pretenem posar en marxa un servei web, però no som especialistes en els components i tecnologies necessaris per construir-lo, ni volem invertir gaire esforç per aconseguir-ho. És possible tenir algun tutorial que ens indiqui, pas a pas, com construir i configurar cada peça; això no obstant, aquesta alternativa no està lliure de complicacions: les instruccions depenen de la data en què es redacten perquè el programari involucrat és molt variat i evoluciona amb el temps. Això fa que els detalls, que en el seu moment encaixaven, puguin no fer-ho actualment.

És un **problema de reproductibilitat** que il·lustrem amb dos exemples:

- Quan en un Dockerfile s'especifica una imatge base, com ara **FROM ubuntu:latest**, la versió de la imatge fa dos anys no és la mateixa que l'actual (ha passat de la sèrie 22.X a la 24.X). La imatge generada **POT NO SER LA MATEIXA**.
- Si la seqüència per generar la imatge inclou alguna cosa semblant a **RUN apt-get upgrade**, l'actualització que pot produir depèn del programari instal·lat i de les versions més recents. El resultat obtingut fa dos anys no coincideix amb el que ara s'obtidria.

Això ens ha de portar a fixar amb precisió els components que hi intervenen i les seves versions, vigilant les possibles actualitzacions perquè segueixin funcionant com s'espera. Aquesta responsabilitat no sembla correspondre amb el nostre paper "no especialista", cosa que ens porta a plantejar-nos com a alternativa alguna solució *prefabricada* que discutim a continuació.

### 4.1. SERVEI WEB BASAT EN IMATGE DE WORDPRESS

Per aquest apartat prendrem un enfocament didàctic amb el punt de partida següent: volem posar en marxa un servidor web basat en WordPress. Com que no tenim cap especificació en contra, implementarem el servei en un equip amb LINUX (gratuït), de manera que complim rigorosament el requisit LAMP de WordPress. Podem esbrinar com instal·lar i configurar aquest sistema com a suma de les peces individuals, però segur que hi ha alternatives més senzilles i còmodes.

Els nostres primers passos solen consistir a consultar a la web, emprant **wordpress** com a terme principal, destacant <https://es.wordpress.org/>. A més trobem informació sobre allotjament, aplicacions autoinstal·lables, llibres, documentació, ...massa<sup>[1]</sup> referències.

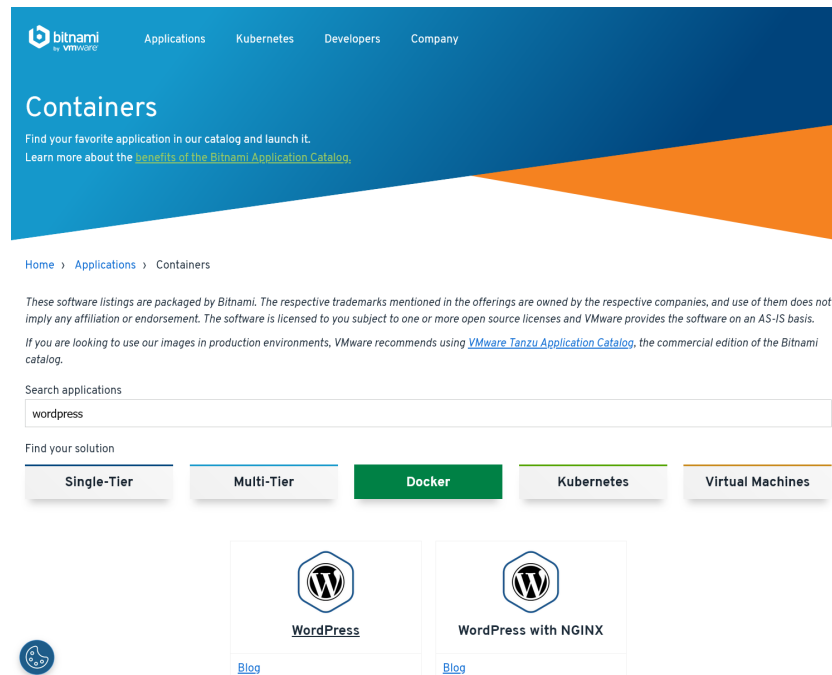
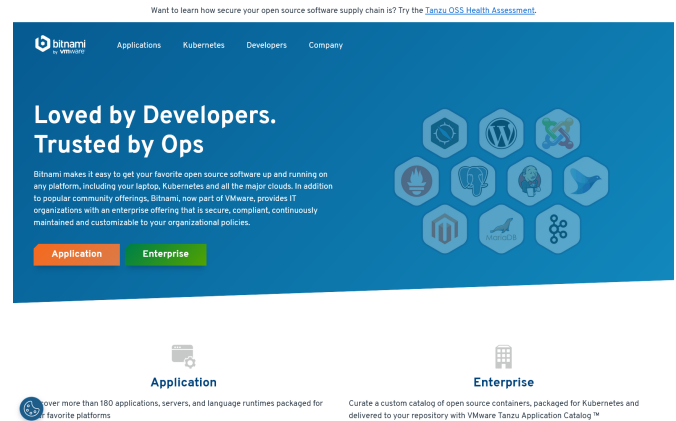
Ja que ens trobem en el tema de desplegament de serveis, és oportú aplicar les tecnologies que esmentem aquest cas, i enfocar el problema des d'una altra perspectiva: un contenidor (o diversos interconnectats) per implementar un servei amb WordPress? Ara el centre de les informacions serà Docker, i la cerca "docker

wordpress" ja *solo* torna 62.200.000 de resultats (un 4.64% de la cerca anterior). La primera referència que apareix és la de la imatge oficial ([https://hub.docker.com/\\_/wordpress](https://hub.docker.com/_/wordpress)). En general tota aquesta web té un elevat interès si es vol localitzar imatges per a contenidors, però prendrem una altra alternativa menys coneguda perquè il·lustra altres possibilitats: <https://bitnami.com/>


Bitnami és una empresa propietat de VMware, una veterana especialista en virtualització. Disposa d'un catàleg d'aplicacions/serveis, la majoria gratuïts, adaptats per a diversos tipus de virtualització, núvols, contenidors o execució en màquines natives.

El catàleg d'aplicacions consta de més de 180 entrades, i seleccionant la pestanya Docker també s'observen centenars d'elements.

Buscant WordPress apareixen dues entrades: ens interessa la que **no** esmenta nginx.







**Bitnami package for WordPress**  
[www.wordpress.org](https://www.wordpress.org)

Updated  
about 23 hours ago
Version  
6.7.0

DEPLOYMENT OFFERING

On the cloud

Containers

On my computer

Single-Tier
Multi-Tier
**Docker**
Kubernetes
Virtual Machines

**BITNAMI PACKAGE FOR WORDPRESS 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](#).

**ADDITIONAL RESOURCES**


- [Readme](#)
- [Configuration](#)
- [Other guides](#)
- [Why use Bitnami Containers?](#)

CONTAINERS



Available versions

6.7.0-0



Dockerhub Debian

**INSTALLATION**

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

**PREREQUISITE**

- Docker Engine 1.10.0

Les instruccions per a la instal·lació del contenidor es troben a <https://github.com/bitnami/containers/tree/main/bitnami/wordpress#how-to-use-this-image>, i es poden resumir en...

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

El fitxer `docker-compose.yml` (apareix com `docker-compose_wordpress.yml` al material subministrat) és:

```
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

```

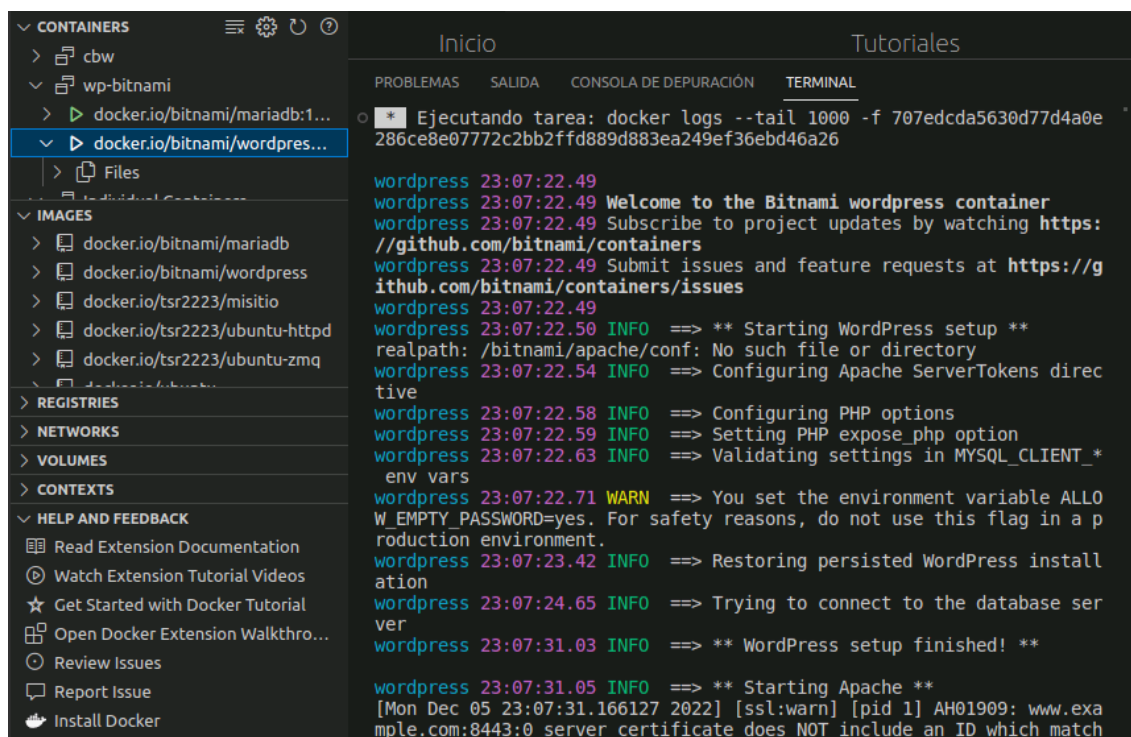
De manera que, amb el que sabem de Docker, podem predir què passarà la primera vegada que executem l'ordre `docker compose up -d`, no...?

```

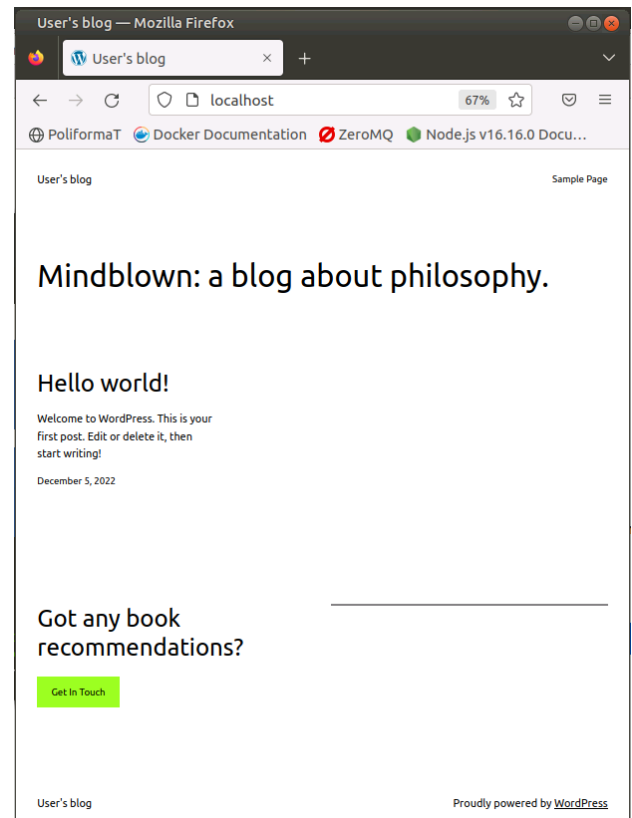
[+] Running 5/5
  wordpress Pulled 29.1s
  75d78dd64cc0 Pull completi 27.7s
  mariadb Pulled 21.0s
  9b0425129f68 Pull completi 6.3s
  a165d97ad6b6 Pull completi 19.5s
[+] Running 5/5
  Network wp-bitnami_default Cre... 0.2s
  Volum "wp-bitnami_mariadb_data" Created 0.0s
  Volum "wp-bitnami_wordpress_data"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

```

Pots veure diagnòstics molt més detallats des de Visual Studio Code amb **View Logs** aplicat, p.ex., al contenidor `docker.io/bitnami/wordpress-1`



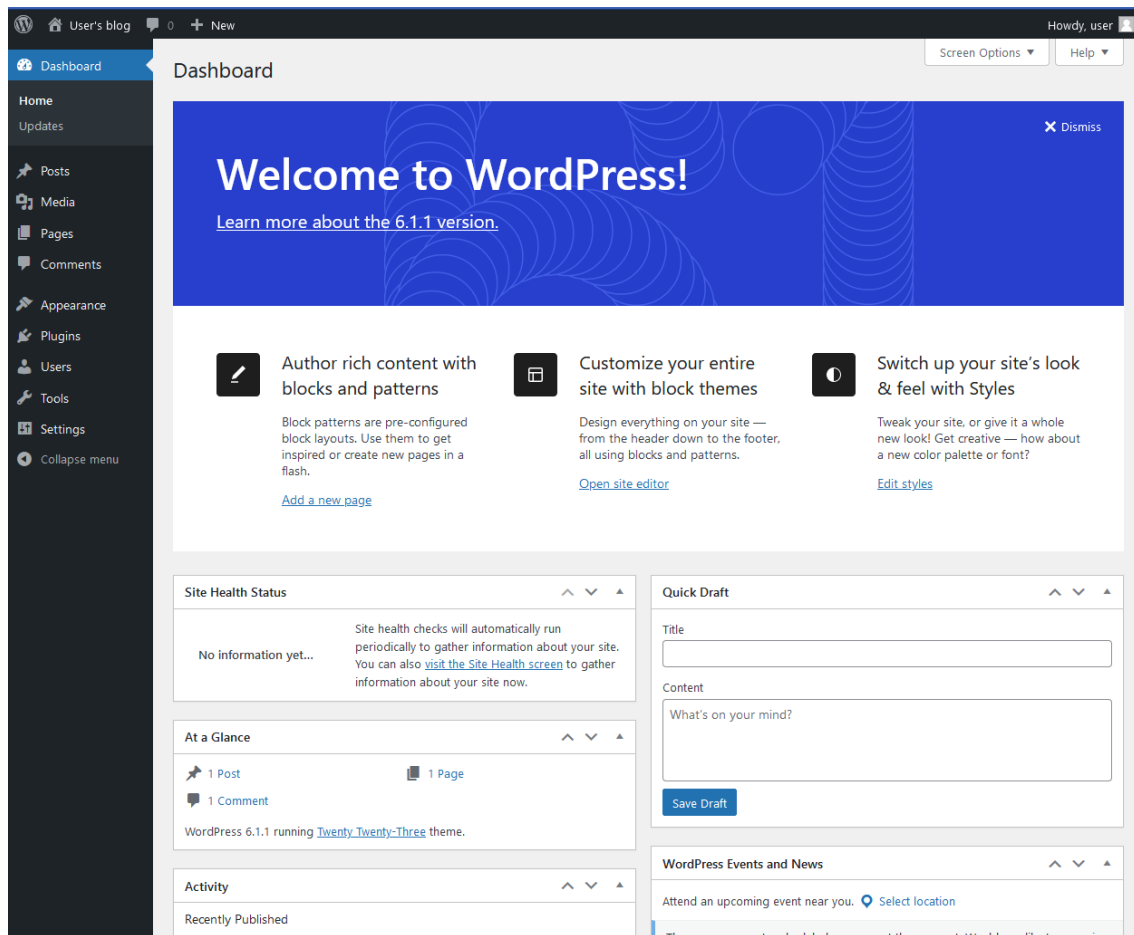
Accedim a aquest nou servidor des del navegador web del nostre amfitrió mitjançant la URL <http://localhost/>, o des d'un altre equip amb accés al portal mitjançant 'http://tsr-milogin-2425.dsicv.upv.es/'. La pàgina resultant no té una gran aparença, com s'observa a la dreta.



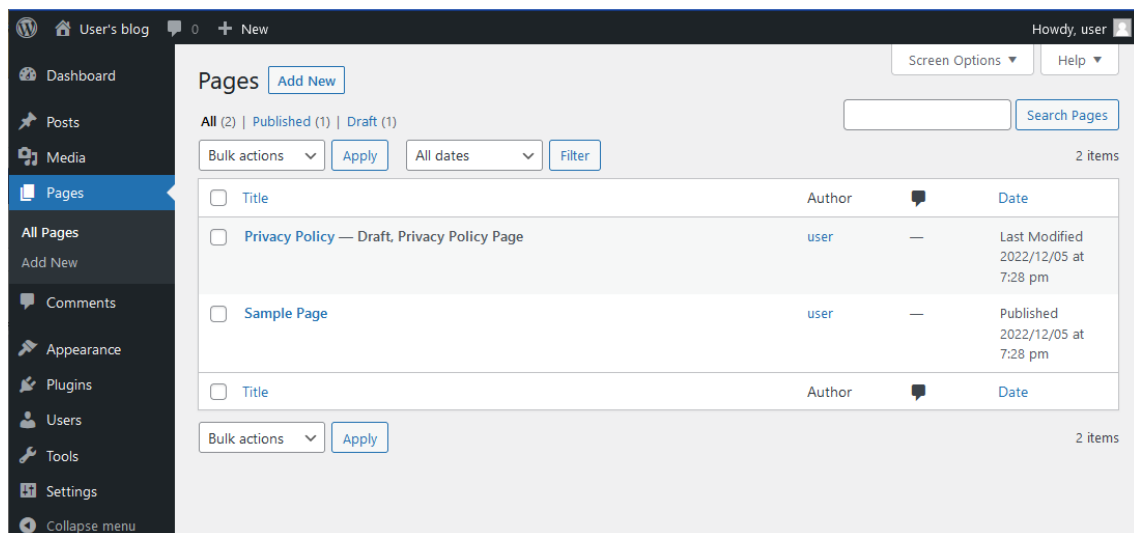
## Qüestió

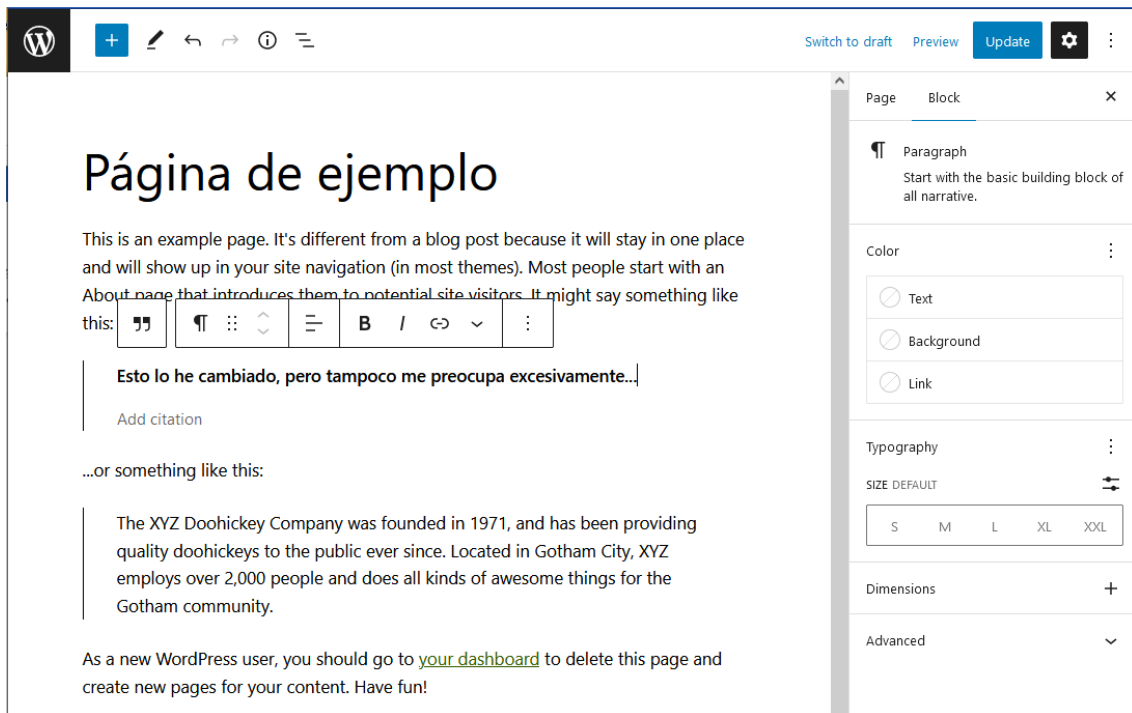
He pogut esbrinar que, dins de l'amfitrió, la URL <http://192.168.2.19:8080/> és equivalent a <http://localhost/>. Com ho he pogut esbrinar? Reconstrueix els passos necessaris per fer-ho al teu virtual de portal.

WordPress permet accedir a la instal·lació en **mode edició**, però exigeix que l'usuari s'identifiqui prèviament. Usant la URI `/admin` i amb les credencials `user/bitnami` podem passar a administrar els continguts del servidor mitjançant el *Dashboard*.



En aquest punt fem una activitat senzilla: canviar la pàgina d'exemple (enllaçada amb *Sample Page*). Per això fem servir l'opció *Pages* a la columna esquerra, i premem sobre *Sample Page*. A continuació en modifiquem el contingut.





Premem *Update* perquè el canvi s'apliqui, i ja podem accedir al contingut de la forma habitual (p.ex. amb <http://localhost/>), polsar a l'enllaç a dalt a la dreta (ha canviat el títol) i verificar la nova informació.



El nostre objectiu NO és utilitzar WordPress ni crear continguts. No inverteixis gaire temps a rematar detalls d'aparença.

Un cop realitzada aquesta acció, aturem el servei (**docker compose down**) i després el tornem a llançar (**docker compose up -d**). Què creus que passarà?

```
> 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 informació i configuració que hem modificat encara *segueix aquí*.

## Qüestió

Examina el fitxer **docker-compose.yml** per determinar a quin lloc de l'amfitrió es guarden els continguts de la base de dades. En absència d'un lloc específic, examina **/var/lib/docker/volumes** (pot requerir privilegis mitjançant **sudo**).

## Qüestió final

Podeu observar que el sistema desplegat és un sistema LAMP compost per dues peces diferenciabls. Recordeu que en el primer tema, a propòsit de la Wikipedia, es va esmentar la possibilitat d'escalat separant el servidor APACHE d'un altre dedicat al SGBD. Al sistema WordPress contemplat en aquest apartat es pot aplicar la mateixa separació, col·locant el servei **wordpress** en un contenidor d'un equip, i la BD **mariadb** en un altre (hauries de col·laborar amb un/a company/a), de forma que el contenidor *wordpress*

actui com a client extern de *mariabd*. Pots *esbossar* els passos necessaris per aconseguir-ho? No es pretén un desplegament simultani.

[1] Google, amb data 20 de novembre de 2024, informa sobre 1.340.000.000 resultats

## 5. ANNEXOS

### 5.1. PREVI: CONSTRUIR LA IMATGE TSR-ZMQ

Aquesta imatge és el punt de partida per als servidors del portal. Hem de construir-la, en cas que encara no s'hagi fet, seguint els passos de l'apartat 2.1.

### 5.2. ANNEX 1: CBW (BÀSIC)

Mantenim sense canvis el codi dels 3 components. Els Dockerfile de client i treballador estan parametritzats i així es facilita la resolució de dependències.

#### 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"
```

Quan els Dockerfile estiguin preparats, desplegueu amb `docker compose up`

#### 5.2.2. CLIENT I EL SEU DOCKERFILE

##### client.js

```
1 const {zmq, lineaOrdnes, traza, error, adios, conecta} = require('../tsr')
2 lineaOrdnes("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"))

```

Al vostre Dockerfile només canviem l'última línia

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

Construir amb **docker build**

### 5.2.3. WORKER I EL SEU DOCKERFILE

**worker.js**

```

1 const {zmq, lineaOrdnes, traza, error, adios, conecta} = require('../tsr')
2 lineaOrdnes("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 procesaPetición(cliente, separador, mensaje) {
11     traza('procesaPetición', 'cliente separador mensaje', [cliente, separador,
12     mensaje])
13     setTimeout(()=>{req.send([cliente, '', `${mensaje} ${id}`]}), 1000)
14 }
15 req.on('message', procesaPetición)
16 req.on('error', (msg) => {error(`${msg}`)})
17 process.on('SIGINT', adios([req], "abortado con CTRL-C"))

```

Al vostre Dockerfile només canviem l'última línia

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

Construir amb **docker build**. Ja es pot desplegar i provar

## 5.3. ANNEX 2: CBWL (AMB LOGGER)

Aquest annex proporciona informació sobre el logger i el desplegament de CBW que s'ajusta a aquest nou cas.



### 5.3.1. BROKERL I EL SEU DOCKERFILE

Només necessitem incorporar tres coses al codi de **brokerl**:

1. El nou parell IP/port del logger, com a darrers arguments

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

2. El socket de comunicació PUSH, i la connexió amb el logger

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

3. I seleccionar els moments en què enviar la informació al logger. En aquest exemple s'aplica a l'entrada als listeners del broker per als dos routers

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

### 5.3.2. LOGGER I EL SEU 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')})
```

Al vostre Dockerfile s'han de complir els requisits en amfitrió: directori preexistent amb 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

Tot allò relacionat amb clients i treballadors roman igual, però hem de donar entrada al nou servei logger, ja les noves dependències perquè...

- El broker pot connectar amb el logger (`$LOGGER_HOST` i `$LOGGER_PORT`)
- El logger pugui conèixer el directori de treball cedit per l'amfitrió (`$LOGGER_DIR`)

A continuació es mostra el contingut de `docker-compose.yml`, del qual interessa especialment els serveis `bro` i `log`

```
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: 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
```