TSR - PRÀCTICA 3
CURS 2017/18

DESPLEGAMENT DE SERVEIS

AQUESTA ÉS UNA VERSIÓ PROVISIONAL DE L'ENUNCIAT DE LA PRÀCTICA 3, QUE SERÀ AMPLIADA EN ELS PRÒXIMS DIES.

El laboratori 3 es desenvoluparà en tres sessions. Els seus objectius principals són:

> "Que l'estudiant comprenga alguns dels reptes que comporta el desplegament d'un servei multi-component, presentant-li un exemple d'eines i aproximacions que pot emprar per a abordar aquests reptes"

Aquesta pràctica depèn fortament de l'anterior, **ØMQ**, requerint d'ella:

- Un coneixement exhaustiu del sistema client-broker-worker (amb socket ROUTER-ROUTER en el broker).
- La capacitat de modificar el codi d'aquests components per a adaptar-lo a canvis en la seua especificació, destacant:
    - el mode "verbose",
    - la incorporació de batec
    - i el suport per a múltiples tipus de treball.

A més necessitaràs refrescar els teus coneixements sobre signatura digital per a un escenari en el qual es necessita atorgar autenticitat a les comunicacions, amb una infraestructura mínima de claus, i signant missatges perquè el broker admeta la seua circulació.

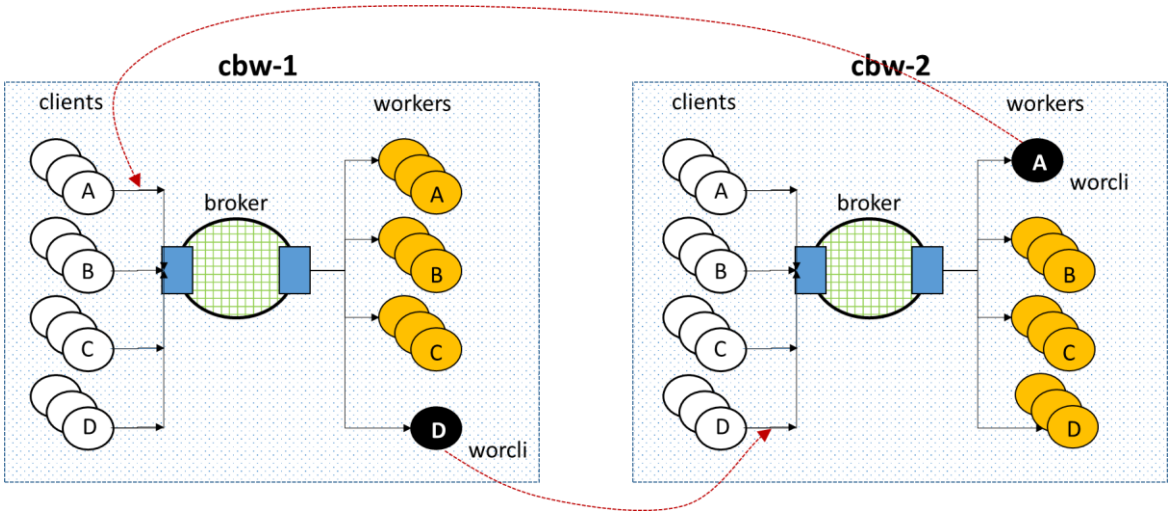
En aquest document, la **part 1** es refereix a l'ús bàsic de Docker, eina que usarem com a base de la nostra estratègia de desplegament. Es limita a referenciar algunes activitats descrites en el seminari 4 com a presa de contacte. L'activitat amb la qual conclou aquest apartat és el desplegament d'un servei *client-broker-worker* (***cbw***).

La **part 2** pren ***cbw*** com a base i proposa, en primer lloc[1], aplicar els canvis de la pràctica 2 per a suportar mode "verbose", batec i classes de treball. En segon lloc se sol·licita una modificació perquè els missatges de mode "verbose" dels treballadors es dirigisquen a un nou component

[1] No és un començament difícil si en la pràctica 2 es van cobrir els objectius i s'ha conclòs l'apartat 1 d'aquesta pràctica 3

(*logger*) que, més tard, pot ser consultat. En tercer lloc[2] se sol·licita la incorporació d'autenticació dels missatges de clients i treballadors que passen pel broker.

La **part 3** pretén afegir al sistema anterior un nou tipus de component (***worcli***) que actue alhora com a treballador d'un broker i com a client d'un altre. Per tant es tracta d'un escenari amb 2 sistemes cbw "connectats". Els criteris pels quals intervé worcli son determinants per a entendre el seu funcionament. Aquest nou sistema ofereix, entre altres coses, la possibilitat de comunicar sistemes cbw que es troben en servidors virtuals diferents.



Per a aquesta pràctica es necessita:

1. Els materials accessibles en PoliformaT (tsr_lab3_material.zip) en el directori corresponent a la tercera pràctica.
2. La màquina virtual assignada al principi de curs, que conté ja una instal·lació de Docker i permet realitzar els exercicis plantejats en aquesta pràctica.
   - En cas d'emprar un compte d'usuari *normal*, que no té privilegis d'administració, no serà possible interactuar amb el dimoni docker. Per a permetre-ho s'haurà d'executar:

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

   …. que afig a l'usuari invocador al grup docker. Hauràs de tornar a iniciar sessió perquè tinga efecte.

[2] Aquesta subpart ha sigut **cancel·lada**

CONTINGUT

0 INTRODUCCIÓ

Els serveis són el resultat de l'execució d'una o diverses instàncies de cadascun dels components de programari emprats per a 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 dels components siga repetible de forma determinista, i que l'execució de les instàncies de components s'aïlle 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.
4. Finalment, el servei obtingut i desplegat ha de ser capaç d'escalar els seus recursos en funció de la demanda, aspecte que haurà de ser considerat tant en l'especificació de la configuració d'aquest desplegament, com en el mecanisme que execute un escalat estàtic (iniciar amb un nombre d'instàncies de cada component) reconstruint els enllaços adequats.
    - L'opció dinàmica és més complexa perquè requereix crear o destruir instàncies per demanda, sense perjudicar la disponibilitat del servei. Abordar aquesta funcionalitat requeriria més esforç del que disposem en aquestes pràctiques.

Una gran part dels conceptes que es posen en joc en aquesta pràctica tenen la seua base en l'escenari descrit en l'apartat 4.2.1 del seminari Tecnologies per al Desplegament, encara que també intervenen altres condicionants pràctics que no poden ser ignorats.

En aquest laboratori explorem formes de construir components, configurar-los, connectar-los i executar-los per a formar aplicacions distribuïdes escalables d'una *forma raonablement* senzilla. Per a fer això ens dotem de tecnologias especialitzades en aquest àmbit.

1. Ens enfrontem al primer problema amb l'ajuda del *framework* **Docker**. Tal com s'ha estudiat en el seminari, **Docker** ens proveeix d'eines per a preparar de forma reproduïble tota la pila de programari necessària per a la instanciació d'un component, de manera que totes les seues dependències de programari les tinga disponibles sense interferir en les dependències d'altres components.

   A més, aquesta pila de programari es deixa preparada perquè l'execució d'un component (la seua instanciació) succeïsca dins d'un contenidor Linux. Els contenidors Linux permeten "aïllar" l'execució de tasques de manera que el que succeeix dins d'un contenidor no afecte al que succeeix dins d'un altre contenidor, contribuint de manera apreciable a que la instanciació d'un component es puga fer tantes vegades com es vulga.

2. Per a resoldre adequadament el segon problema es necessita especificar amb metadades la configurabilitat de cada component, de manera que siga intel·ligible per

un programa que duga a terme les accions oportunes. Donada la nostra elecció de tecnologia (**Docker**) haurem d'entendre com donar a conèixer les dependències perquè el *framework* de Docker les resolga. Concretament necessitarem conèixer com emplenar un ***Dockerfile*** i com referenciar algunes de les informacions contingudes en ell.

3. Per a abordar la tercera necessitat descrita procedirem incrementalment.
    a. Inicialment desplegarem tots els components d'un servei manualment, usant directament ordres docker, usant directament les informacions de configuració necessàries com a paràmetres de les ordres docker.
    b. Posteriorment automatitzarem aquesta activitat mitjançant l'ús de la utilitat **docker-compose**, que estableix una forma d'especificar les relacions entre components per a desplegar una aplicació distribuïda. Això ens obligarà a entendre els fonaments d'aquest nou programa, i l'especificació necessària[3] per a construir el fitxer **docker-compose.yml** amb la interrelació entre els components de la nostra aplicació distribuïda.
4. Finalitzant la nostra relació de solucions a necessitats parcials, l'escalat forma part de la funcionalitat oferida per **docker-compose**, de manera que l'aspecte instrumental és molt més senzill d'abordar. Per això, arribat aquest punt, la dificultat principal es troba en l'adequació dels components de l'aplicació distribuïda per al seu escalat. Es desitja que l'escalat contribuïsca a millorar el rendiment del servei implementat, la qual cosa no pot ser reduïda a un assumpte únicament aritmètic.

Els exercicis concrets que s'exposen utilitzen aplicacions realitzades per nosaltres (codi en NodeJS, amb mòduls) que podem modificar per aconseguir els nostres objectius.

[3] Mitjançant el llenguatge de marques YAML

1 PART 1: PRIMERS PASSOS AMB DOCKER EN EL SEMINARI 4

1.1 Desplegament del servidor de web

En l'exemple d'ús de Docker de l'apartat **3.3.1** de la guia de l'alumne del **seminari 4** es descriu el desplegament d'un servidor de web en 12 passos, on els 7 primers tenen caràcter interactiu (tsr1718/centos-httpd) i els següents es basen en un Dockerfile (imatge misitio).

- Has de realitzar aquesta mateixa activitat en la teua màquina virtual, comprovant amb un navegador (en la màquina virtual) que pots accedir al servei, i que et retorna una pàgina com en la il·lustració anterior.

1.2 Construcció de la imatge per a components amb NodeJS i ZMQ

En l'apartat **3.3.2** de la guia del **seminari 4** es descriuen els passos necessaris per a construir una imatge amb suport per a executar aplicacions NodeJS que requerisquen del middleware ZMQ. Observaràs que se citen els requisits necessaris, però al final de l'apartat es descarta el resultat perquè …

el més productiu és dividir en dues etapes la creació de la imatge:

- *una primera que instal·larà NodeJS (tsr1718/centos-nodejs),*
- *i una altra segona que afegirà ZMQ a l'anterior (tsr1718/centos-zmq).*

No obstant això, observaràs que en el seminari no s'indiquen les ordres necessàries per a construir la primera (centos-nodejs) de les dues imatges, ni com a partir de la primera es pot obtenir la segona (centos-zmq).

- Es pot esbrinar amb molta facilitat a partir de l'apartat 3.3.2

A continuació es mostra el Dockerfile per a centos-nodejs:

```
FROM centos:7.4.1708
RUN curl --silent --location https://rpm.nodesource.com/setup_6.x | bash -
RUN yum install -y nodejs
```

- Ordre de construcció: docker build -t tsr1718/centos-nodejs .

Anàlogament, per a centos-zmq:

```
FROM tsr1718/centos-nodejs
RUN yum install -y epel-release
RUN yum install -y zeromq-devel make python gcc-c++
RUN npm install zmq
```

- Ordre de construcció: docker build -t tsr1718/centos-zmq .

Aquestes dues imatges seran preses com a base per a construir el codi dels components:

- TCPProxy (es basa en centos-nodejs)
- client/broker/worker (es basen en centos-zmq)

1.3 Encaixant TCPProxy amb el servidor de web

L'apartat **4.2.2** de la guia de l'alumne del seminari 4 esmenta com construir un component desplegable a partir de TCPProxy, però deixa sense resoldre els arguments d'invocació de Proxy.js atès que depenen del servidor de web desplegat al començament d'aquesta PART 1.

Posa en funcionament el servidor, esbrina la IP del contenidor i ajusta l'ordre CMD que has de col·locar en el Dockerfile de TCPProxy.

- Executa el component. Què has d'usar com a URL en el navegador (sobre la virtual del portal) per a accedir al servidor a través del proxy?

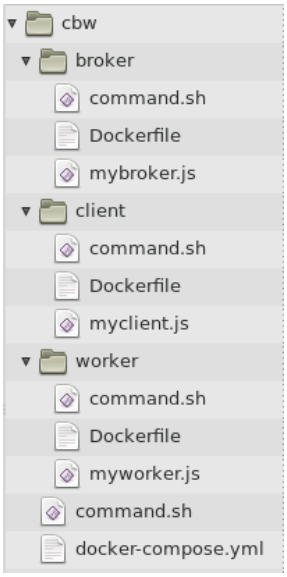
1.4 Desplegament de les imatges individuals de client/broker/worker

El mateix problema que apareix en el desplegament anterior (TCPProxy al costat del servidor de web) es pot trobar en l'apartat **4.2.1** de la guia de l'alumne del seminari 4.

Has de generar les imatges dels tres components seguint els passos descrits, excepte la creació de centos-zmq que ja ha sigut realitzada. Cada component ha de comptar amb una carpeta amb el seu propi Dockerfile i els arxius que necessite.

Obri 5 finestres. Executa aquestes instruccions per a comprovar que tot encaixa:

- Finestra 1: docker run broker
- Finestra 2: docker run worker
- Finestra 3: docker run worker
- Finestra 4: docker run client
- Finestra 5: docker run client



Desgraciadament el codi original de myclient.js solament emet una petició. Canvia-ho perquè realitze 10 sol·licituds abans de finalitzar. Aquest nou codi ha d'anomenar-se my10client.js, però la imatge seguirà amb el mateix nom: client. **Desplega-la**.

1.5 Unint múltiples components en una aplicació distribuïda

L'apartat **6.4** de la guia de l'alumne del seminari 4 esmenta com construir un desplegament orquestrat de diversos components per a crear una aplicació (**cbw**) distribuïda. Has de seguir aquells passos canviant myclient.js per my10client.js.

Com a ajuda et mostrem com ha de quedar el Dockerfile del client:

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./my10client.js /zmq/myclient.js
WORKDIR /zmq
# We assume that each client is linked to the broker
# container.
CMD node myclient $BROKER_URL
```

Si has col·locat la configuració del desplegament de l'apartat 6.4 del seminari, per a executar 8 clients i 3 treballadors usarem:

```
docker-compose up --scale cli=8 --scale wor=3
```

2 PART 2: AMPLIACIÓ DE CLIENT-BROKER-WORKER

2.1 Adaptant cbw: newclient, newbroker i newworker

Com recordaràs, al llarg de la pràctica 2 es proposen modificacions sobre el codi d'alguns dels components que formen el sistema **cbw**. De tots ells seleccionem la unió d'aquestes 3 modificacions:

- Suport per al **mode *verbose*** (que al seu torn inclou suport per a arguments d'invocació), descrit en l'apartat 1.2.4 i en l'annex 2 de la pràctica 2.
- Modificació per a permetre **tipus de treball**, especificats per clients, admesos per treballadors, i suportats pel broker. Aquest aspecte va ser descrit i resolt en l'apartat 3.1 de la pràctica 2.
- Incorporació de **batec** (*heartbeat*) perquè el broker detecte (i recupere) fallades de workers, descrit i resolt en l'apartat 3.2 de la pràctica 2.

Has d'aconseguir modificar el codi original dels components de cbw per a incloure totes les característiques anteriors, juntament amb les 10 peticions per client.

- Si no s'han arribat a completar tots els canvis, llavors la resta de la pràctica queda **degradada**. En aquest cas pren, almenys, com a codi de partida el de myclient1.js[4], myworker1.js i mybroker1.js, de l'apartat 3.1.2 de la pràctica 2, modificats per a admetre en la seua invocació els paràmetres necessaris.
- S'ha d'incloure el suport per a 4 tipus de petició (classID en el codi del broker): 'A ','B', 'C' i 'D'. Aquests 4 valors han de ser arguments per al broker, tal com es va comentar en el butlletí anterior.

Des d'aquest moment anomenem newclient1, newbroker1 i newworker1 a les versions modificades per a incorporar les 3 característiques anteriors (o la seua versió degradada).

2.2 Anotant els diagnòstics

En la seua forma actual, el mode *verbose* pot ser incòmode perquè obliga a llegir els missatges en pantalla en el moment en què es produeixen. És una pràctica estesa que, excepte urgència, aquests diagnòstics s'acumulen seqüencialment en algun arxiu per a la seua posterior consulta; de fet fins i tot existeixen formats reconeguts per a aquestes anotacions que permeten a aplicacions externes *digerir* aquesta informació. No és el nostre cas.

Podríem triar que cada component guarde les seues anotacions en un fitxer, però no és còmode comptar amb moltes fonts d'informació.

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

[4] Realment serà my10client1.js

En resum: afegim un nou component al que tots els processos en mode verbose envien les informacions que abans escrivien en pantalla. Aquest *logger* anirà agregant a un arxiu (cbwlog.txt) cada missatge rebut.

Col·locarem un directori específic (cbw_log) per a organitzar aquest apartat.

Aspectes destacables:

- És important que aquest arxiu cbwlog.txt no perda el seu contingut entre invocacions. Recorda que la naturalesa efímera dels contenidors és ací un problema a resoldre. Necessitaràs usar un volum Docker per a connectar aquest fitxer amb un espai de l'amfitrió.
- Si els components han 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 ZMQ aplicable: PULL per a *logger* i PUSH per als *seus usuaris*[5] serà suficient (encara que altres variacions permetrien altres característiques).

2.2.1 logger.js

El seu codi és senzill per a qui ja ha experimentat ZMQ:

```
01: // logger in NodeJS
02: // First argument is port number for incoming messages
03: // Second argument is file path for appending log entries
04:
05: var fs = require('fs');
06: var zmq = require('zmq')
07: ,   log = zmq.socket('pull')
08: ,   args = process.argv.slice(2);
09:
10: var loggerPort = args[0] || '8066';
11: var filename = args[1] || "/tmp/myfile.log";
12:
13: log.bindSync('tcp://*:'+loggerPort);
14:
15: log.on('message', function(text) {
16:   fs.appendFileSync(filename, text);
17: })
```

En el nostre propòsit per substituir les invocacions de console.log per enviaments de missatges a *logger*, optarem pel següent:

- Importar el mòdul util
- Establir un socket (anomenat log) de tipus PUSH connectat al *logger*
- Substituir on siga necessari console.log(*arguments*) per log.send(util.format(*arguments*))
    - Aquesta instrucció es pot dividir en 2, utilitzant una variable auxiliar, per a crear sentències més clares

Pots veure l'efecte en el codi de my10client_vb.js que es mostra en l'apartat següent.

També el seu Dockerfile, pràcticament idèntic a uns altres ja estudiats:

[5] client, broker o worker

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./logger.js /zmq/logger.js
WORKDIR /zmq
EXPOSE 8066
CMD node logger 8066
```

2.2.2 Modificacions en el codi dels components

Cada vegada que incorporem noves exigències o funcionalitats al nostre codi, es produeix un increment en la seua complexitat. És una cosa difícilment evitable, però pot ser suavitzada amb alguna mesura simplificadora.

- En aquest cas el mode ***verbose*** **SEMPRE ESTÀ ACTIU**, en lloc de triar-ho en la invocació. Aquest supòsit simplifica la gestió dels arguments d'invocació dels components.
- Es manté la necessitat d'un argument per a establir connexió amb *el logger*, sempre especificat en últim lloc (args[args.length-1]).

Prenent com a referència un client que produeix 10 peticions de la classe B, ja modificat per a suportar el mode verbose (newclient_vb.js), el seu codi resultant seria:

```
01: // newclient_vB in NodeJS , implicit verbose activation mode
02: //  - v stands for verbose mode
03: //  - B stands for classID="B"
04:
05: var zmq = require('zmq')
06: , requester = zmq.socket('req')
07: , util = require('util');
08:
09: var nMsgs=10;
10: var args = process.argv.slice(2);
11: var loggerURL = args[args.length-1];
12: var log       = zmq.socket('push');
13: args.pop(); // rest of argument processing will follow
14: var diag;
15:
16: var brokerURL = args[0] || 'tcp://localhost:8059';
17: var myID = args[1] || 'NONE';
18: var myMsg = args[2] || 'Hello';
19: var classID = 'B';
20:
21: if (myID != 'NONE')
22:   requester.identity = myID;
23: requester.connect(brokerURL);
24: log.connect(loggerURL);
25: diag = util.format('Client (%s) with class "%s" connected to %s', myID, classID,
    brokerURL);
26: log.send(diag);
27:
28: requester.on('message', function(msg) {
29:   diag = util.format('Client (%s) has received reply "%s"', myID, msg.toString());
30:   log.send(diag);
31:   if (--nMsgs == 0)
32:     process.exit(0);
33:   else
34:     requester.send([myMsg,classID]);
35: });
36: diag = util.format('Client (%s) sending request "%s" of class "%s"', myID, myMsg,
    classID);
37: log.send(diag);
38:
39: requester.send([myMsg,classID]);
```

2.2.3 Noves dependències dels components

Tots ells, incloent el broker, necessitaran conèixer com connectar amb *logger*. Aquesta situació és similar a les relacionades entre client (o treballador) i broker, i suposa la necessitat de col·locar una variable d'entorn a sustituir en el desplegament. Per exemple, el Dockerfile per al client serà:

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./newclient _vB.js /zmq/myclient.js
WORKDIR /zmq
# We assume that each client is linked to the broker
# container.
CMD node myclient $BROKER_URL $LOGGER_URL
```

2.2.4 Desplegament de logger

És necessària una consideració que no ens havia preocupat fins ara: com es relaciona el fitxer d'anotacions (/tmp/myfile.log) amb el sistema de fitxers de l'amfitrió? Mitjançant una secció volumes[6] en la descripció del desplegament.

Suposa que ja hem creat l'arxiu /tmp/logger.log en l'amfitrió

Si únicament desitgem desplegar aquest component, i no tota l'aplicació distribuïda, haurem d'emprar una invocació de docker run amb una opció equivalent a la secció volumes

docker run –v /tmp/logger.log:/tmp/myfile.log *paràmetres*

2.2.5 Desplegament conjunt del nou servei cbw_log

En el contingut de l'arxiu de configuració del desplegament de l'aplicació cbw_log ressaltem les modificacions respecte al desplegament sense *logger*.

```
version: '2'
services:
  wor:
    image: worker
    build: ./worker/
    links:
      - bro
      - log
    environment:
      - BROKER_URL=tcp://bro:8059
      - LOGGER_URL=tcp://log:8066
  cli:
    image: client
    build: ./client/
    links:
      - bro
      - log
    environment:
      - BROKER_URL=tcp://bro:8060
      - LOGGER_URL=tcp://log:8066
  bro:
```

[6] disposes d'informació addicional en el material de referència del seminari 4

```
    image: broker
    build: ./broker/
    links:
      - log
    environment:
      - LOGGER_URL=tcp://log:8066
  log:
    image: logger
    build: ./logger/
    volumes:
      - /tmp/logger.log:/tmp/myfile.log
```

L'engegada amb docker-compose no té cap novetat. És interessant que, des de l'amfitrió, pot accedir-se a les anotacions mitjançant /tmp/logger.log.

Prova a realitzar: el desplegament bàsic requereix un arxiu d'anotacions buit, i l'execució d'una combinació composta per 4 clients de tipus B, 2 treballadors de tipus B, 1 broker i 1 logger.

Qüestió: reflexiona, sense necessitat d'executar, què ocorreria si intentem els següents escenaris:

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

3 ANNEX: RECURSOS NECESSARIS PER A AQUESTA PRÀCTICA

1. Ports oberts:

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

  En executar firewall-cmd --list-all, en la secció public, ha d'aparèixer una línia ports: 8000-8100/tcp