TSR - PRÀCTICA 3
CURS 2017/18

DESPLEGAMENT DE SERVEIS

El laboratori 3 es desenvoluparà al llarg de 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 tals reptes"

Aquesta pràctica depèn fortament de l'anterior, **ØMQ**, requerint de la mateixa:

- 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-ho a canvis en la seua especificació, destacant:
    - la manera 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.

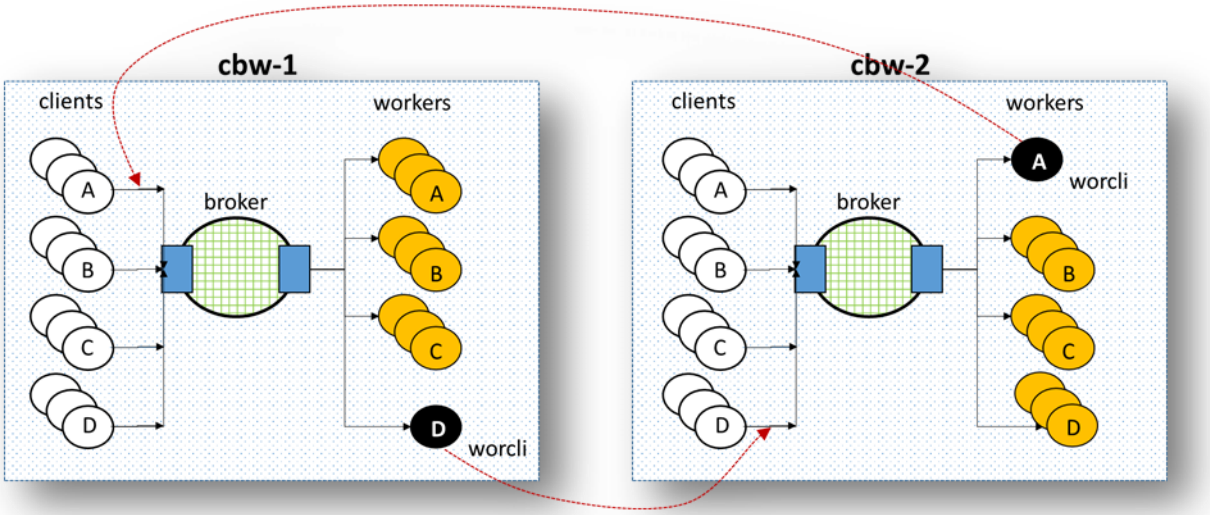
En aquest document, la **part 1** es refereix al maneig bàsic de Docker, eina que usarem com a base de la nostra estratègia de desplegament. Es limita a referenciar algunes activitats descrites en el seminari 4 com a presa de contacte. L'activitat amb la qual conclou aquest apartat és el desplegament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

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

[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

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

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 són 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. Pots comparar el seu contingut amb la imatge de l'últim annex.
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 compta amb privilegis d'administració, no serà possible interactuar amb el dimoni docker. Per a permetre-ho s'haurà d'executar:

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

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

CONTINGUTS

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 seua instanciació 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 tecnologies 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 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 puga repetir-se sense problemes.

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 a 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 radica 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 reduït 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 a 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 el de 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?

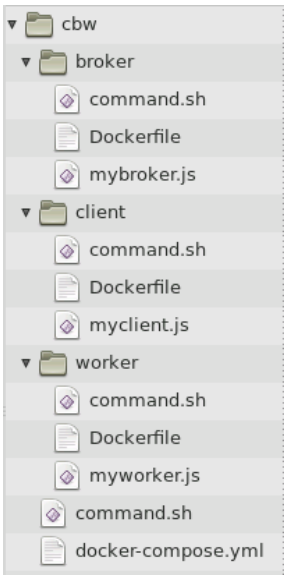
1.4 Desplegament de les imatges individuals de client/broker/worker

El mateix problema que apareix en el desplegament anterior (TCPProxy al costat de 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 face 10 sol·licituds abans de finalitzar. Aquest nou codi ha d'anomenar-se my10client.js, però la imatge seguirà mantenint el 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: ENRIQUINT 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 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[5] 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 en cas d'urgència, els 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* la 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. Si pretenguérem que tots els components anotaren directament els seus diagnòstics en un únic fitxer centralitzat, estaríem completament fora de lloc… un sistema distribuït amb un fitxer compartit?. **Targeta roja directa**.

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

[5] Aquest aspecte solament és una proposta d'ampliació final

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 el 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*[6] 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:

```
1:  // logger in NodeJS
2:  // First argument is port number for incoming messages
3:  // Second argument is file path for appending log entries
4:
5:  var fs = require('fs');
6:  var zmq = require('zmq')
7:  ,   log = zmq.socket('pull')
8:  ,   args = process.argv.slice(2);
9:
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:

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./logger.js /zmq/logger.js
WORKDIR /zmq
```

[6] client, broker o worker

```
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.

- Én aquest cas el mode ***verbose*** **SEMPRE ESTÀ ACTIU**., en lloc de triar-ho en la invocació. Això 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:

```
1:  // newclient_vB in NodeJS, implicit verbose activation mode
2:  //  - v stands for verbose mode
3:  //  - B stands for classID="B"
4:
5:  var zmq = require('zmq')
6:  , requester = zmq.socket('req')
7:  , util = require('util');
8:
9:  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 la ja relacionada entre client i treballador amb el broker, i suposa la necessitat de col·locar una variable d'entorn a susituir 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[7] en la descipción 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:
    image: broker
    build: ./broker/
    links:
      - log
    environment:
      - LOGGER_URL=tcp://log:8066
  log:
    image: logger
    build: ./logger/
    volumes:
      - /tmp/logger.log:/tmp/myfile.log
```

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

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 desplegar-ho en els següents escenaris:

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

3 PART 3: CAP A UNA APLICACIÓ DISTRIBUÏDA MÉS COMPLETA

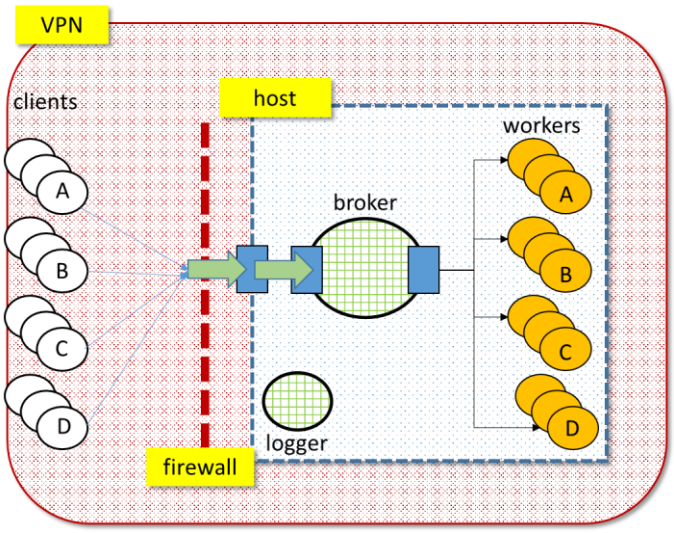
El model bàsic **cbw** amb els seus enriquiments segueix trobant-se distanciat dels problemes tècnics que poden trobar-se en una aplicació distribuïda. En aquest apartat pretenem incorporar alguns detalls que ens porten en aquesta direcció.

El primer pas que prenem ens facilita la reducció parcial de la complexitat de les classes de treball; el segon es refereix a la reformulació del paper que juguen els clients en aquest esquema, i com s'habilita l'accés des de l'exterior de l'amfitrió; el tercer cobreix l'addició d'un nou component (**worcli**) que aprofita la preparació anterior.

Aquests objectius no són especialment ambiciosos, però sí suficients per als objectius ajustats als recursos que disposem.

3.1 Redefinint servei i clients

Els clients no són part de l'aplicació distribuïda, per la qual cosa hauran d'interactuar amb el servei mitjançant algun punt ben conegut per a poder encarregar-li treball. Problemes que apareixen:



Quin és l'endpoint del servei? L'URL del bróker.

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

Tots dos problemes poden ser resolts reservant un port de l'amfitrió que es farà correspondre amb la IP i port del bróker en el desplegament.

- Si el tallafocs de l'amfitrió (*host*) es troba configurat correctament, la línia `ports` de `docker-compose.yml` farà el que volem.
- Els equips del portal en TSR ja estan configurats per a permetre l'accés des de l'exterior als ports 8000 a 8100. En cas contrari executarem:

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

Ara podrem disposar de clients externs que connectaran amb el servei mitjançant un URL fix `tcp://hostIP:8059` que conduirà les peticions al bróker.

- Els clients mantenen **requisits** per a la seua execució (NodeJS + ZMQ) que **no es compleixen en tots els equips**.

    - En les virtuals d'escriptori dels laboratoris ja es disposa de NodeJS, encara que sabem que l'ús de ports pot ser conflictiu si s'usen ordres com bind o listen.
        - Les de tipus connect, com en clients i treballadors, NO plantegen problemes.
    - L'única **manca és el suport de ZMQ**, que s'aconsegueix copiant en el vostre directori la carpeta **node_modules** inclosa en el material d'aquesta pràctica.
- Els clients externs a l'amfitrió no poden interactuar amb el logger.
    - Tampoc sembla aconsellable oferir externament aquest servei.
- Els equips fora de la VPN han de, a més, obtenir accés a ella.

Pots realitzar una prova mínima de funcionament amb la versió de cbw que desitges, però considera aquests passos:

1. El desplegament en el portal només inclou broker i treballadors (opcionalment també el logger).
2. Has adaptat la configuració de desplegament per a afegir en la secció del broker:

```
ports:
"8060:8060"
```

3. En l'equip d'escriptori executaràs el/els client/s, i ja has afegit la carpeta node_modules.
4. També en aquest equip esbrinaràs la IP del teu servidor en el portal (p. ex. amb una ordre ping tsr-8888-1718.dsic.cloud que ens retorna 192.168.105.111)
    - 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 es dirigisquen al mateix port del broker.

Suposem aquesta versió del client (**newclient_external.js**) en l'equip d'escriptori:

```
1:  // newclient_external in NodeJS, implicit verbose activation mode
2:  // brokerURL and classID must be provided as parameters
3:  // node newclient_external brokerURL classID
4:
5:  var zmq = require('zmq')
6:  , requester = zmq.socket('req');
7:
8:  var nMsgs=10;
9:  var args = process.argv.slice(2);
10: if (args.length < 2) {
11:   console.log ("Usage: node newclient_external brokerURL classID");
12:   process.exit(-1);
13: }
14: var classID = args.pop();  // rest of argument processing will follow
15:
16: var brokerURL = args[0];
17: var myID = args[1] || 'NONE';
18: var myMsg = args[2] || 'Hello';
19:
20: if (myID != 'NONE')
21:   requester.identity = myID;
22: requester.connect(brokerURL);
23: console.log('Client (%s) with class "%s" connected to %s', myID, classID, brokerURL);
24:
25: requester.on('message', function(msg) {
26:   console.log('Client (%s) has received reply "%s"', myID, msg.toString());
27:   if (--nMsgs == 0)
28:     process.exit(0);
29:   else {
30:     console.log('Client (%s) sending request "%s" of class "%s"', myID, myMsg, classID);
31:     requester.send([myMsg,classID]);
```

```
32:   }
33: });
34:
35: console.log('Client (%s) sending first request "%s" of class "%s"', myID, myMsg, classID);
36: requester.send([myMsg,classID]);
```

5. Prenent les dades de l'exemple, i per al cas particular de sol·licituds del tipus B, hauríem d'invocar-ho d'aquesta forma:

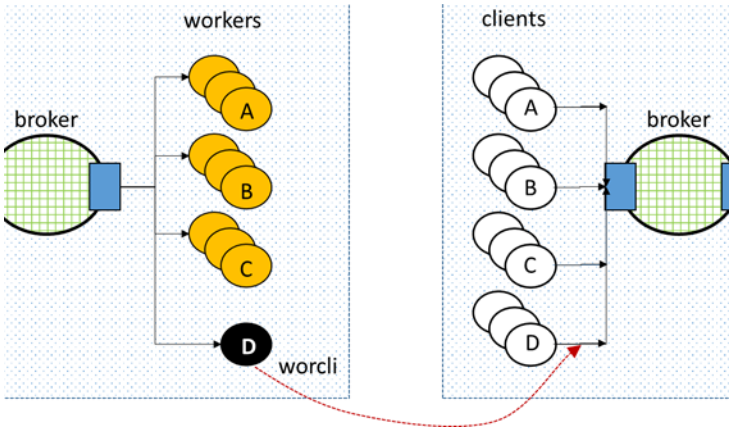
```
node newclient_external tcp://192.168.105.111:8060 B
```

6. I en l'amfitrió arrancar el servei (broker i workers) mitjançant docker-compose up.

Amb aquestes proves i les modificacions necessàries estem començant la nostra preparació per a construir components que interactuen encara que no es troben en el mateix amfitrió.

3.2 Encadenant diversos sistemes cbw: el worcli

El worcli és un nou component el codi del qual prové de la fusió dels rols worker i client per a un mateix tipus de treball T.



La idea és que aquest component redirigisca peticions d'un bróker bk1, que disposa de treballadors del tipus T, a un altre, bk2, que disposa de clients per al tipus de treball T.

- Per a fer això es connecta amb bk1 com a treballador, i envia la petició com a client a bk2. El camí invers requereix recordar en worcli l'identificador del client que es trobava en el missatge de petició.

Per a la seua execució requerirà 4 paràmetres: l'URL de bk1, l'URL de bk2, el retard que afig i el tipus (classID) de treball que ha de processar.

Codi de worcli

```
1:  // worcli
2:  // invoked with "node worcli bk1URL bk2URL [myID [connText]] delay classID"
3:  // 4 parameters llaure mandatory
4:  var zmq = require('zmq')
5:  , responder = zmq.socket('req')
6:  , requester = zmq.socket('req');
7:
8:  var args = process.argv.slice(2);
9:  if (args.length < 4) {
10:   console.log ("Usage: node worcli bk1URL bk2URL [myID [connText]] delay classID");
11:   console.log ("Redirects bk1's classID requests to bk2 broker, increasing delay ms");
12:   process.exit(-1);
13: }
14: var classID = args.pop();
15: var delay = parseInt(args.pop()); // network delay, in ms
16: var beURL = args[0]; // worcli behaving like a worker for bk1
17: var feURL = args[1]; // worcli behaving like a client for bk2
18: var myID = args[2] || 'NONE';
```

```
19: var connText = args[3] || 'worcli'; // marking initial offering
20: var pendingClient;
21:
22: if (myID != 'NONE') {
23:   responder.identity = myID;
24:   requester.identity = myID;
25: }
26: responder.connect(beURL);
27: requester.connect(feURL);
28:
29: responder.on('message', function(client, delimiter, msg) {
30:   pendingClient = client; // only one waiting client, so we don't need a queue
31:   setTimeOut(function(){
32:     requester.send(msg);}, delay/2); // 50% forwarding
33: });
34:
35: requester.on('message', function(msg) {
36:   setTimeOut(function(){
37:     responder.send([pendingClient,''].concat(msg));}, delay/2); // 50% returning
38: });
39:
40: responder.send([connText,classID]);
```

En aquest apartat 3.2, la major part de les propostes són **preguntes obertes** que conviden a explorar les possibilitats, limitacions i implicacions de cadascuna. Totes prenen com a punt de partida una situació amb 2 amfitrions (*hosts*) en el portal, cadascun amb un desplegament que inclou 1 broker (accessible pel port 8060 del seu amfitrió), i múltiples treballadors.

3.2.1 Pregunta 1

Pot el *worcli* executar-se en els equips d'escriptori, de la mateixa manera que un client extern? Si observem que ha de cobrir 2 rols, prompte podem reformular la pregunta com **pot actuar com a worker fora de l'amfitrió?**

3.2.2 Pregunta 2

Discuteix quines possibilitats hi ha perquè worcli puga actuar com a worker en l'equip d'escriptori. Si no et convenç cap, **hauràs d'executar-lo en un dels amfitrions**: en quin dels dos?, de què depèn?

3.2.3 Pregunta 3

Si *el worcli* s'executa en un amfitrió, pot **formar part d'un desplegament** del servei (mitjançant un docker-compose.yml)? Com a alternativa, podries posar-ho en funcionament "***a mà***"? Argumenta la resposta a ambdues preguntes. Si la segona és afirmativa, explica com ho faries.

3.2.4 Pregunta 4

Suposa que alguna de les alternatives ens permet enganxar els sistemes mitjançant aquest *worcli*.

- Imaginem que s'especialitza en la classe de treballs D , sense competència per part de cap altre worker d'aquest amfitrió.
- Imagina, a més, que emprem la implementació amb suport de batec en el broker, amb tots els detalls que això comporta.

Argumenta **si hi haurà diferències importants** (i quines són) entre les 3 invocacions següents, i quines repercussions tindrà quan el broker allotjat en el primer amfitrió (192.168.105.111) reba alguna petició d'un client de tipus D:

```
node worcli tcp://192.168.105.111:8060 tcp://192.168.105.112:8059 200 D
```

```
node worcli tcp://192.168.105.111:8060 tcp://192.168.105.112:8059 1200 D
```

```
node worcli tcp://192.168.105.111:8060 tcp://192.168.105.112:8059 2200 D
```

(El codi d'invocació mostrat solament és una il·lustració que **no ha de prendre's com a pista** o resposta, excepte el canvi en el retard especificat en cada cas i la seua relació amb el batec)

3.2.5 Pregunta 5

Al final de l'apartat 2.1 s'indica, parlant dels 4 tipus de petició, que "*Aquests 4 valors han de ser arguments per al broker*", però no han sigut tractats com a tals en cap moment d'aquest butlletí. Això es deu al fet que la forma en la qual el broker obté aquests valors pot merèixer un estudi a part: la flexibilitat del broker depèn en part de la seua independència respecte al nombre i noms dels tipus de petició.

- L'apartat 3.1.2 de la pràctica 2 ja esmentava alguns aspectes relacionats.

El broker pot obtenir els valors **estàticament**, a l'inici, sense possibilitat de modificació al llarg de la seua execució. Dins de les possibles implementacions per a aquesta alternativa es pot trobar la definició dins del seu propi codi (és el nostre cas), l'obtenció des d'arguments d'invocació (és la indicació original) o des d'un fitxer de configuració.

No obstant això, la major flexibilitat es podria obtenir mitjançant un comportament adaptatiu i, per tant, **dinàmic**. Per exemple, en la pràctica 1 apareixia la figura del programador (i controlador) que interactuava amb el proxy per a reprogramar les adreces i ports. Ací es podria fer alguna cosa similar per a establir els tipus de treball i canviar-los al llarg de l'execució del broker.

- Quins són els possibles problemes que es poden trobar si s'elimina una classe de treball mentre el sistema està funcionant?

Com a alternativa es podria dissenyar un broker en el qual els tipus de petició s'anaren descobrint a mesura que els missatges passen per ell. En aquesta alternativa mai s'eliminen tipus i no es necessita un component configurador addicional.

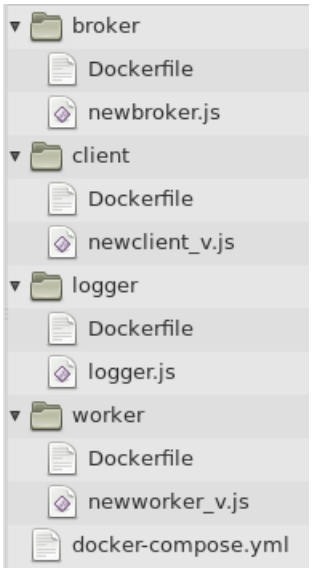
- Pots proporcionar una implementació aproximada del broker? Pots reflexionar sobre l'adaptació del **docker-compose.yml** per a un cas de nombre indeterminat de tipus de treball?

4 ANNEX: CBW AMB TIPUS DE TREBALL, BATEC I LOGGER

Si abordem el desplegament de treballadors i clients prenent cada classe de treball com un element diferencial, necessitarem 2x4 = 8 casos. És fàcil entendre que això pot millorar-se, llevat que desitgem una *"explosió"* d'entrades en l'arxiu de configuració del desplegament i en l'estructura de directoris que requereix. Com pot abordar-se?

1. Quant al **fitxer de configuració del desplegament**, si desitgem un control fi sobre el nombre d'instàncies de cada tipus, ens obliguem a poder referenciar-les per separat en l'ordre docker-compose run, com a argument que acompanya a l'opció --scale. Dit d'una altra forma, no podem controlar aquests components si no els diferenciem en la configuració del desplegament, reconeixent la impossibilitat de simplificació en aquest aspecte.
2. No obstant això, quant a l'estructura de directoris sí que podem reduir la seua complexitat a canvi de modificar el codi dels components client i worker: la idea és subministrar el tipus de treball (classID) com **un paràmetre d'invocació que pot ser establit en el desplegament** (com l'URL del broker).
    - D'aquesta forma comptaríem amb un únic codi i Dockerfile per a tots els clients, i un altre parell per a tots els treballadors.

La incorporació de batec no modifica l'estructura en absolut, i el component *logger* tampoc necessita canvis.



4.1 Docker-compose.yml (*abreujat*)

```
version: '2'
services:
  worA:
    image: wk
    build: ./worker/
    links:
      - bro
      - log
    environment:
      - BROKER_URL=tcp://bro:8059
      - LOGGER_URL=tcp://log:8066
      - CLASSID=A
# same structure for rest of workers: worB, worC and worD
  cliA:
    image: cl
    build: ./client/
    links:
      - bro
      - log
    environment:
      - BROKER_URL=tcp://bro:8060
      - LOGGER_URL=tcp://log:8066
      - CLASSID=A
# same structure for rest of clients: cliB, clic and cliD
  bro:
    image: broker
    build: ./broker/
    links:
      - log
    environment:
      - LOGGER_URL=tcp://log:8066
  log:
```

```
image: logger
build: ./logger/
volumes:
  - /tmp/logger.log:/tmp/cbwlog.txt
```

4.2 client

Es tracta de la versió executable i desplegable en l'amfitrió.

newclient_v.js

```
1:  // newclient_v in NodeJS, implicit verbose activation mode
2:  //  - v stands for verbose mode
3:  // classID must be provided as a parameter
4:  // CMD node myclient $BROKER_URL $CLASSID $LOGGER_URL
5:
6:  var zmq = require('zmq')
7:  , requester = zmq.socket('req')
8:  , util = require('util');
9:
10: var nMsgs=10;
11: var args = process.argv.slice(2);
12: var loggerURL = args.pop();
13: var log       = zmq.socket('push');
14: var diag;
15: var classID = args.pop();  // rest of argument processing will follow
16:
17: var brokerURL = args[0] || 'tcp://localhost:8059';
18: var myID = args[1] || 'NONE';
19: var myMsg = args[2] || 'Hello';
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]);
```

Dockerfile

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

4.3 broker

newbroker_hb.js

```
1:  // ROUTER-ROUTER request-reply broker in NodeJS.
2:  // Work classes. Logger variant
3:  // Worker availability-aware variant.
4:  //
5:  // As code grows, complexity increases. This version returns to the
6:  // original structure with auxiliar functions (sendToWorker & sendRequest)
7:
8:  var zmq      = require('zmq')
9:  , frontend = zmq.socket('router')
10: , backend  = zmq.socket('router')
11: , util = require('util');
12:
13: var args = process.argv.slice(2);
14: var loggerURL = args[args.length-1];
15: var log       = zmq.socket('push');
16: args.pop(); // rest of argument processing will follow
17: var diag;
18:
19: var fePortNbr = args[0] || 8059;
20: var bePortNbr = args[1] || 8060;
21: var workers = [];
22: var clients = [];
23:
24: const classIDs = ['A','B','C','D'] // Do we really need this limitation?
25: for (var i in classIDs) {
26:   workers[classIDs[i]]=[];
27:   clients[classIDs[i]]=[];
28: }
29:
30: const answerInterval = 2000;
31: var busyWorkers = [];
32:
33: function showMessage(t, action, contents) { // adapted from auxfunctions
34:  log.send(util.format('Broker (%s) %s this message:', t, action));
35:  var msg = Array.apply(null, contents);
36:   msg.forEach( (value,index) => {
37:      log.send(util.format('     Segment %d: %s', index, value));
38:   })
39: }
40:
41: frontend.bindSync('tcp://*:'+fePortNbr);
42: backend.bindSync('tcp://*:'+bePortNbr);
43: log.connect(loggerURL);
44: diag = util.format('Broker listening on fePort %d and bePort %d', fePortNbr, bePortNbr);
45: log.send(diag);
46:
47: // Send a message to a worker.
48: function sendToWorker(msg, classID) {
49:   var myWorker = msg[0];
50:   diag = util.format('Broker passing client (%s) request to queued worker (%s) through backend.',
    msg[2], msg[0]);
51:   log.send(diag);
52:   showMessage('be', 'sending', msg);
53:   backend.send(msg);
54:   busyWorkers[myWorker] = {};
55:   busyWorkers[myWorker].classID = classID;
56:   busyWorkers[myWorker].msg = msg.slice(2);
57:   busyWorkers[myWorker].timeout =
58:     setTimeout(generateTimeoutHandler(myWorker),answerInterval);
59: }
60:
61: // Function that sends a message to a worker, or
62: // holds the message if no worker is available now.
63: // Parameter 'args' is an array of message segments.
64: function sendRequest(args, classID) {
65:   if (workers[classID].length > 0) {
66:     var myWorker = workers[classID].shift();
67:     var m = [myWorker,''].concat(args);
68:     sendToWorker(m, classID);
69:   } else {
70:     diag = util.format('Broker queueing client (%s) (cl%s queue length: %d).', args[0], classID,
    clients[classID].length+1);
71:     log.send(diag);
72:     clients[classID].push( {id: args[0],msg: args.slice(2)});
73:   }
74: }
75:
```

```
76:  function generateTimeoutHandler(workerID) {
77:  diag = util.format('Broker "installing" function for handling worker (%s) timeout.', workerID);
78:  log.send(diag);
79:    return function() {
80:      diag = util.format('Broker "running" function for worker (%s) timeout. busyWorkers.length=%d',
     workerID, busyWorkers.length);
81:      log.send(diag);
82:      var msg = busyWorkers[workerID].msg;
83:      var classID = busyWorkers[workerID].classID;
84:      delete busyWorkers[workerID];
85:      sendRequest(msg, classID);
86:    }
87:  }
88:
89:  frontend.on('message', function() {
90:    var args = Array.apply(null, arguments);
91:    var classID = args.pop();
92:    showMessage('fe', 'receiving', arguments);
93:    diag = util.format('Broker receiving frontend request: "%s" from client (%s) with class "%s".',
     args[2], args[0], classID);
94:    log.send(diag);
95:    sendRequest(args, classID);
96:  });
97:
98:  function processPendingClient(workerID, classID) {
99:    if (clients[classID].length>0) {
100:       var nextClient = clients[classID].shift();
101:       var msg = [workerID,'',nextclient.id,''].concat(nextClient.msg);
102:           sendToWorker(msg, classID);
103:       return true;
104:     } else
105:       return false;
106:   }
107:
108:   backend.on('message', function() {
109:   showMessage('be', 'receiving', arguments);
110:     var args = Array.apply(null, arguments);
111:     var classID = args.pop();
112:     if (args.length == 3) {
113:   diag = util.format('Broker receiving backend request: "%s" from worker (%s) with class "%s".',
     args[2], args[0], classID);
114:   log.send(diag);
115:       if (!processPendingClient(args[0],classID)) {
116:   diag = util.format('Broker queueing worker (%s) (wk%s queue length: %d).', args[0], classID,
     workers[classID].length+1);
117:   log.send(diag);
118:            workers[classID].push(args[0]);
119:       }
120:     } else {
121:       var workerID = args[0];
122:       diag = util.format('Broker receiving reply: "%s" from worker (%s)', args[4], workerID);
123:       log.send(diag);
124:       clearTimeout(busyWorkers[workerID].timeout);
125:       args = args.slice(2);
126:       diag = util.format('Broker passing worker (%s) reply to client (%s) through frontend.', workerID,
     args[0]);
127:       log.send(diag);
128:       showMessage('fe', 'sending', args);
129:       frontend.send(args);
130:       if (!processPendingClient(workerID,classID)) {
131:           diag = util.format('Broker queueing worker (%s) (wk%s queue length: %d).', args[0], classID,
     workers[classID].length+1);
132:           log.send(diag);
133:           workers[classID].push(workerID);
134:       }
135:     }
136:   });
```

Dockerfile

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./newbroker_hb.js /zmq/mybroker.js
WORKDIR /zmq
EXPOSE 8059 8060
CMD node mybroker $LOGGER_URL
```

4.4 worker

newworker_v.js

```
1:  // newworker_v server in NodeJS,  implicit verbose activation mode
2:  //  - v stands for verbose mode
3:  // classID must be provided as a parameter
4:  // CMD node myworker $BROKER_URL $CLASSID $LOGGER_URL
5:
6:  var zmq = require('zmq')
7:  , requester = zmq.socket('req')
8:  , util = require('util');
9:
10: var nMsgs=10;
11: var args = process.argv.slice(2);
12: var loggerURL = args.pop();
13: var log       = zmq.socket('push');
14: var diag;
15: var classID = args.pop();  // rest of argument processing will follow
16:
17: var backendURL = args[0] || 'tcp://localhost:8060';
18: var myID = args[1] || 'NONE';
19: var connText = args[2] || 'id';
20: var replyText = args[3] || 'World';
21:
22: if (myID != 'NONE')
23:   responder.identity = myID;
24: responder.connect(backendURL);
25: log.connect(loggerURL);
26: diag = util.format('Worker (%s) with class "%s" connected to %s', myID, classID, backendURL);
27: log.send(diag);
28:
29: responder.on('message', function(client, delimiter, msg) {
30:   diag = util.format('Worker (%s) has received request "%s" from client "%s"', myID, msg.toString(),
    client);
31:   log.send(diag);
32:   setTimeout(function() {
33:       diag = util.format('Worker (%s) sending "%s" back to broker for client "%s"', myID, replyText,
    client);
34:       log.send(diag);
35:       responder.send([client,'',replyText, classID]);
36:   }, 1000);
37: });
38: diag = util.format('Worker (%s) communicating availability for class "%s"', myID, classID);
39: log.send(diag);
40:
41: responder.send([connText,classID]);
```

Dockerfile

```
FROM tsr1718/centos-zmq
RUN mkdir /zmq
COPY ./newworker_v.js /zmq/myworker.js
WORKDIR /zmq
# We assume that each worker is linked to the broker
# container.
CMD node myworker $BROKER_URL $CLASSID $LOGGER_URL
```