TSR - PRÀCTICA 2
CURS 2017/18

ØMQ

L'examen d'aquesta pràctica es farà el dia 11 de desembre

Aquesta pràctica, que consta de quatre sessions, té com a propòsit:

"Entrenar a l'alumne en l'ús del sistema de missatgeria ØMQ. En finalitzar aquesta part de laboratori, l'alumne ha de conèixer com desenvolupar aplicacions amb ØMQ sobre NodeJS"

Per a complir amb aquest propòsit, establim objectius segons el següent guió:

Els alumnes aprofundiran en els patrons bàsics de comunicació en ØMQ, que ja han vist en les classes de seminari, fent diverses modificacions en programes donats que implementen aquests patrons.

Els alumnes implementaran un **repartidor de missatges de petició de servei**:

- En aquest sistema, hi haurà un servidor (anomenat ***proxy***) que exposa un ***socket* ØMQ** del tipus **ROUTER**, acceptant connexions remotes de clients.
- Addicionalment, el servidor tindrà connexions amb un ***pool*** de treballadors disposats a gestionar les peticions dels clients. S'ha d'usar la millor manera d'establir aquestes connexions d'entre les possibles facilitades per ØMQ.
- Quan arriba un missatge pel socket d'entrada, el *broker* selecciona un dels treballadors, i li passa el missatge.
- Quan el treballador acaba de gestionar el missatge, li envia un missatge de resposta al servidor, qui ho reenvia de tornada a qui li va enviar la petició original.

Finalment, els alumnes dissenyaran i implementaran algunes modificacions en aquest sistema, que és l'objectiu avaluable d'aquesta pràctica.

En aquest butlletí, l'apartat **1** descriu les activitats relatives al primer objectiu (estudi dels patrons bàsics de comunicació en ØMQ), l'apartat **2** aborda el segon objectiu (estudi, implementació i xicotetes modificacions del repartidor de missatges de petició de servei), i l'apartat **3** proposa i descriu modificacions importants a fer sobre el sistema implementat en l'apartat anterior. També s'afigen annexos amb informació d'utilitat.

Es recomana emmagatzemar el material produït durant aquesta pràctica en una carpeta, **TSRPrac2**.

El laboratori presenta una configuració específica per a TSR des de la pràctica de NodeJS, resumida com "*una màquina virtual per a cada alumn@, accessible a través de la VPN de portal-ng.dsic.cloud*". Com ja saps, aquesta organització ens permet afrontar les limitacions que imposa la coexistència de sessions[1] sobre els mateixos equips, destacant les restriccions en el control de recursos (ports especialment).

[1] És la forma dominant de treball en els laboratoris del DSIC

CONTINGUT

0 INTRODUCCIÓ A ØMQ (REPÀS)

En les activitats de laboratori s'utilitzarà el *middleware* de comunicacions basat en cues, anomenat **ØMQ** o **ZeroMQ**, versió 4.1.x. Els seus binaris ja es troben instal·lats. Per a poder usar-ho amb el nostre llenguatge de programació asincrònica, **NodeJS**, s'han instal·lat els *bindings* corresponents a aquest llenguatge amb l'ordre (en el directori $HOME de l'usuari creat en la nostra màquina virtual):

```
bash-4.1$ npm install zmq
```

L'execució correcta d'aquesta ordre ja ha generat un directori `$HOME`[2]`/node_modules/zmq` que conté tot el necessari per a usar ØMQ en programes de NodeJS.

Aleshores, els programes que incloguen o requerisquen ØMQ…

```
var zmq = require('zmq');
```

…no tindran problemes per a accedir a la biblioteca de ØMQ. Si, en executar un programa que incloga l'anterior línia de codi, es genera un missatge d'error similar al següent…

```
module.js:340
    throw err;
          ^
Error: Cannot find module 'zmq'
    at Function.Module._resolveFilename (module.js:338:15)
    at Function.Module._load (module.js:280:25)
    at Module.require (module.js:364:17)
    …
```

…és senyal que no s'ha executat correctament l'ordre `npm install zmq`.

Es disposa d'informació completa sobre ØMQ en *el seu website* oficial: http://zeromq.org. En particular, quant a estudi i documentació, són essencials:

- "**ØMQ – The** Guide ", disponible en http://zguide.zeromq.org/page:all
- "**The ØMQ Reference Manual**", disponible en http://api.zeromq.org

En aquesta documentació, C és el llenguatge de programació usat com a referència. A més, també s'inclouen alguns exemples programats en altres llenguatges (C++, Java, Haskell, NodeJS, Python, etc.).

En "ØMQ – The Guide " s'informa de l'existència d'un dipòsit Git públic que conté tots els exemples de la guia. Qualsevol pot clonar aquest dipòsit mitjançant l'execució de `git clone`.

- En el directori corresponent a la segona pràctica de TSR en els laboratoris (`/asigDSIC/ETSINF/tsr/lab2`), hem col·locat un arxiu **zmq_nodejs_examples.tgz** que conté tots aquests exemples per a NodeJS.
- En les virtuals del portal també s'ha col·locat un directori (`/root/zmq_nodejs_examples`) amb el mateix material.

[2] Si connectes com root, llavors `$HOME = /root`

La execució dels mateixos sense adaptar, pot generar errors d'accés des de l'exterior atès que no tots els ports s'han habilitat. Convé recordar que en la pràctica anterior es va permetre aquest accés únicament als **ports 8000 a 8100**.

En el següent apartat, estudiarem i modificarem alguns d'aquests exemples. Es tracta d'implementacions correctes de diversos patrons bàsics de comunicació amb ØMQ.

1 PROGRAMACIÓ BÀSICA AMB ØMQ

Volem fer diverses modificacions en alguns dels programes d'exemple. Per a fer això, s'ha de crear un nou directori dins de **TSRPrac2**, anomenat ***zmqbasico***, al que s'afegiran els programes exemple que seran objecte de modificació en aquest primer apartat.

1.1 Estudi del patró REQ - REP

Per a començar, anem a estudiar dos programes, els codis dels quals es reprodueixen a continuació, que mostren la interacció més senzilla possible amb *els sockets* ØMQ:

1.1.1 hwclient.js

```
01:  // Hello world client
02:  // Connects REQ socket to tcp://localhost:8055
03:  // Sends "Hello" to server.
04:
05:  var zmq = require('zmq');
06:
07:  // socket to talk to server
08:  console.log("Connecting to hello world server...");
09:  var requester = zmq.socket('req');
10:
11:  var x = 0;
12:  requester.on("message", function(reply) {
13:    console.log("Received reply", x, ": [", reply.toString(), ']');
14:    x += 1;
15:    if (x === 10) {
16:      requester.close();
17:      process.exit(0);
18:    }
19:  });
20:
21:  requester.connect("tcp://localhost:8055");
22:
23:  for (var i = 0; i < 10; i++) {
24:    console.log("Sending request", i, '...');
25:    requester.send("Hello");
26:  }
27:
28:  process.on('SIGINT', function() {
29:    requester.close();
30:  });
```

1.1.2 hwserver.js

```
01:  // Hello world server
02:  // Binds REP socket to tcp://*:8055
03:  // Expects "Hello" from client, replies with "world"
04:
05:  var zmq = require('zmq');
06:
07:  // socket to talk to clients
08:  var responder = zmq.socket('rep');
09:
10:  responder.on('message', function(request) {
11:    console.log("Received request: [", request.toString(), "]");
12:
13:    // do some 'work'
14:    setTimeout(function() {
15:
16:      // send reply back to client.
17:      responder.send("World");
18:    }, 1000);
19:  });
20:
21:  responder.bind('tcp://*:8055', function(err) {
22:    if (err) {
23:      console.log(err);
24:    } else {
25:      console.log("Listening on 8055...");
26:    }
27:  });
28:
29:  process.on('SIGINT', function() {
30:    responder.close();
31:  });
```

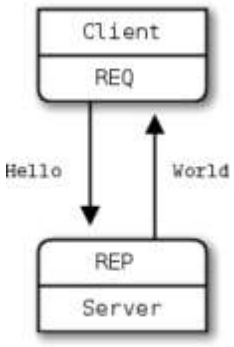

Figura 1: Comunicació client-servidor amb el patró REQ-REP.

Nota: Recorda que has de posar-ho a prova en el teu servidor virtual.

1.1.3 Modificació dels programes: arguments d'invocació

En aquests dos programes s'implementa el patró **REQ** - **REP** de la comunicació amb sockets ØMQ, tal com es mostra en *la figura 1*. Després de llegir i comprendre el seu codi, es faran les següents modificacions en els programes ***hwclient.js*** i ***hwserver.js***:

1) El nou programa **client** tindrà els següents arguments:

   a. URL de l'endpoint en el servidor (per exemple, **tcp://localhost:8000**).
   b. Nombre de peticions a enviar (per exemple, **10**).
   c. Text de la petició a enviar (per exemple, **WORK**).

NOTA: La URL de l'endpoint del servidor conté el protocol de transport utilitzat, l'adreça IP del servidor i el port en el qual aquest atén les peticions, tal com es mostra en l'exemple de l'argument a. del nou client. Utilitzarem aquesta notació per a especificar els *endpoints* en el que resta de pràctica.

2) El nou programa **servidor** tindrà els següents arguments:

   a. Port del servidor (seguint l'exemple posat per a l'endpoint del client podria ser, 8000).
   b. Nombre de segons a esperar en cada resposta (per exemple, **5**).
   c. Text per a la resposta a enviar (per exemple, **DONE**).

Funcionament

- La resposta que el servidor enviarà al client serà la concatenació de la petició del client i el text de resposta pròpia del servidor (el text de l'últim argument en la seua invocació).
- Si qualsevol dels programes no rebera el nombre correcte d'arguments, haurà de mostrar un missatge que informe sobre la seua invocació correcta, i concloure la seua execució.

Proposta de solució

Com sabem, les aplicacions NodeJS poden prendre arguments des de la línia d'ordres. Podem parametritzar els programes perquè les informacions relacionades amb ports de connexió, identificadors i uns altres deixen de representar-se mitjançant constants. De fet, podem emprar una manera *híbrida*, que aprofita l'avaluació lògica amb curtcircuit, en el qual una expressió tal com `A||B` s'avalua a `A` si aquest té valor, o `B` en cas contrari. Així, aplicant això al codi de hwclient :

```
var args = process.argv.slice(2);
var servURL = args[0] || "tcp://localhost:8055";
```

S'interpreta com: col·locar els paràmetres d'invocació en `args`. `servURL` pren com a valor el primer paràmetre, però en la seua absència s'inicialitzarà a `tcp://localhost:8055`.

- Per tant, l'antiga línia 21 ara contindrà `requester.connect(servURL);`

En el codi de `hwserver` apliquem la mateixa estratègia.

L'accés als altres 2 paràmetres en tots dos programes serà similar a la construcció anterior, amb una variable per a cada paràmetre que haurà de ser usada en substitució dels valors constants que apareixen dins del codi.

Els programes obtinguts després d'aquestes modificacions hauràs de nomenar-los com `hwclient_p` i `hwserver_p`.

Proves

Client i servidor haurien de poder executar-se en una mateixa màquina o en dues màquines diferents.

En tots dos casos has d'emprar el servidor virtual:

a) En el primer cas usaràs exclusivament la màquina virtual que et corresponga, i empraràs localhost com IP de servidor.
b) En el segon cas necessitaràs emprar dos virtuals, o bé una virtual i el teu equip d'escriptori local. L'adreça localhost ja no tindrà sentit. Per a triar quin component (client o servidor) col·loques en la virtual i quin en l'equip local, pensa que la virtual té una adreça fixa coneguda de bestreta, mentre que l'equip local pot variar.

El servidor també ha de poder atendre peticiones de diversos clients: observa el comportament del servidor quant a l'ordre d'atenció de peticions concurrents de clients diferents[3].

[3] Recordar el patró bàsic REQ-REP en el cas de múltiples peticionaris, vist al Seminari 3.

1.2 Estudi del patró PUB - SUB

El següent pas és estudiar uns altres dos programes, disponibles en ***zmq_nodejs_examples***, que mostren un altre ús rellevant dels sockets ØMQ:

1.2.1 subscriber.js

```
01:  var zmq = require('zmq')
02:  var subscriber = zmq.socket('sub')
03:
04:  subscriber.on("message", function(reply) {
05:    console.log('Received message: ', reply.toString());
06:  })
07:
08:  subscriber.connect("tcp://localhost:8088")
09:  subscriber.subscribe("")
10:
11:  process.on('SIGINT', function() {
12:    subscriber.close()
13:    console.log('\nclosed')
14:  })
```

1.2.2 publisher.js

```
01:  var zmq = require('zmq')
02:  var publisher = zmq.socket('pub')
03:
04:  publisher.bind('tcp://*:8088', function(err) {
05:    if(err)
06:      console.log(err)
07:    else
08:      console.log("Listening on 8088...")
09:  })
10:
11:  for (var i=1 ; i<10 ; i++)
12:      setTimeout(function() {
13:          console.log('sent');
14:          publisher.send("Hello there!")
15:      }, 1000 * i)
16:
17:  process.on('SIGINT', function() {
18:    publisher.close()
19:    console.log('\nclosed')
20:  })
```

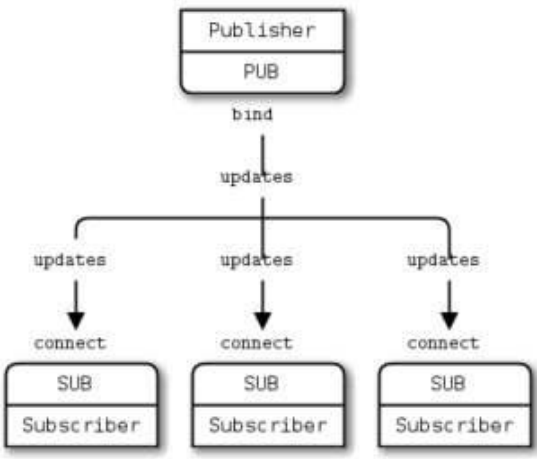

Figura 2: Comunicació publicador-subscriptor utilitzant el patró PUB-SUB

1.2.3 Modificació dels programes: arguments d'invocació

En aquests dos programes s'implementa el patró **PUB** - **SUB** de la comunicació amb sockets ØMQ, tal com es mostra en *la figura 2*. Després de llegir i comprendre el seu codi, es faran les següents modificacions en els programes ***publisher.js*** i ***subscriber.js***.

1) El nou programa **subscriptor** tindrà els següents arguments:

    a. URL de l'endpoint en el publicador (per exemple, **tcp://localhost:8000**).
    b. Descriptor del tipus de missatges al que se subscriu (per exemple, **NOTÍCIES**).

2) El nou programa **publicador** tindrà els següents arguments:

    a. Port del publicador (seguint l'exemple posat per a l'endpoint del client podria ser, **8000**).
    b. Nombre de missatges a publicar (per exemple, **20**).
    c. Primer tipus de missatges a publicar (per exemple, **NOTÍCIES**).
    d. Segon tipus de missatges a publicar (per exemple, **OFERTES**).

Funcionament

- Els subscriptors es connectaran al publicador, especificant la seua URL, així com el descriptor del tipus de missatge que els interesse rebre (filtre de missatges).

- El publicador publicarà el mateix nombre de missatges dels dos tipus especificats. Cada missatge publicat consistirà en la concatenació del descriptor de tipus de missatge i un nombre aleatori.

> **NOTA:** Per a obtenir nombres aleatoris en un rang determinat, pot usar-se la funció ***randNumber***, el codi de la qual es facilita en *l'Annex 1* (*Funcions auxiliars*). Per a fer això podeu afegir la següent ordre d'importació del mòdul ***auxfunctions1718.js*** al principi del programa publicador:
>
> ```
> aux = require("./auxfunctions1718.js");
> ```
>
> i la invocació a la funció serà de la forma `aux.randNumber(…)`.
> (NOTA: El fitxer ***auxfunctions1718.js*** ha de trobar-se en el mateix directori que el publicador).

Proves

Cal comprovar el funcionament del patró PUB-SUB, executant els programes en una mateixa màquina (la teua virtual) i en dues màquines diferents (la teua virtual i el teu equip d'escriptori local). Almenys s'ha de verificar la connexió de dos subscriptors amb diferents filtres de missatges a un mateix publicador.

1.2.4 Modificació dels programes: mode *verbose*

Aquest mode de funcionament ha de permetre obtenir una traça dels punts significatius en l'execució dels programes. En el nostre cas ho són, almenys, l'arribada i eixida de missatges.

La seua incorporació als nostres programes està molt relacionada amb la seua depuració, especialment quan volem observar l'evolució interna de la seua execució. Com no sempre tenim interès en aquesta informació, s'inclourà com a últim argument en la invocació la paraula ***verbose*** quan vulguem activar-ho.

- Observar que aquesta capacitat **complementa a la parametrització** de la invocació.

Així, les línies significatives per a comprovar l'argument seran:

```
var args = process.argv.slice(2);
var verbose = false;
if (args[args.length-1] == "verbose") {
  verbose = true;
  args.pop(); // eliminate only if it appears at the end
}             // rest of argument processing will follow
```

I el seu ús en el codi substitueix les aparicions de

```
    console.log("Listening on 8088...")
```

per unes altres que únicament mostren el missatge en ***mode verbose***:

```
    if (verbose)
      console.log("Listening on 8088...")
```

- Més tard s'esmenta una funció **showMessage** que facilita les operacions per a mostrar el contingut d'un missatge.

La necessitat de depuració és proporcional al grau de complexitat del codi, per la qual cosa en els exemples senzills el mode ***verbose*** sembla sobredimensionat.

En resum, les modificacions necessàries inclouen la parametrització de la invocació i la preparació per a mostrar els missatges interns únicament si apareix com a últim argument la paraula ***verbose***. Als programes obtinguts després d'aquestes modificacions hauràs de nomenar-los com `publisher_pv` i `subscriber_pv`.

1.2.5 Modificació dels programes: publicador rotatori

Sobre `publisher_pv`, sense modificar el subscriptor, es demana fer un canvi que permeta:

- Un nombre i nom de tòpics indeterminat, com arguments d'invocació.
- El publicador `publisher2_pv` ha d'enviar un nombre total de missatges inclòs com a primer paràmetre.

P. ex., una invocació vàlida del publicador seria:

```
node publisher2_pv.js 8088 2 100 moda salut negocis oci verbose
```

Que hauria d'interpretar-se com la publicació d'una seqüència de missatges en la qual s'alterna entre `moda|salut|negocis|oci` cada 2 segons fins a aconseguir els 100 missatges (25 de cadascun)

2 INTERMEDIARI ENTRE CLIENTS I TREBALLADORS

2.1 Aproximació al problema i aspectes clau

L'objectiu d'aquest apartat és la implementació d'un proxy[4] que gestione missatges de petició de servei. En els problemes clàssics de programació concurrent hi ha un model conegut com a **productor-consumidor** que consta d'un espai d'emmagatzematge limitat (*bounded buffer*) per a intercanvi.

- Els processos amb rol productor dipositen (escriuen) en aquest espai un element que els de rol consumidor retiren (consumeixen).

L'estudi d'aquest problema se centra a coordinar (**sincronització condicional**) tots dos tipus de procés sabent que el emmagatzematge es pot omplir o buidar.

- En una situació d'espai buit, els consumidors han de quedar a l'espera de l'arribada d'un nou element.
- En la situació contrària (emmagatzematge ple), són els productors els que han d'esperar fins que es produïsca un buit.

En el cas extrem que l'emmagatzematge intermedi *desapareix*, i totes les operacions s'implementen mitjançant missatges, aquest model de referència s'aproxima més a l'esquema aplicable a aquest apartat.

Amb una mica més de detall, el comportament esperat dels agents que participen és:

- Existeixen processos (**clients**) que sol·liciten serveis. Per a fer-ho construeixen (produeixen) un missatge que han d'enviar a algun servidor (treballador). Si n'hi ha algun disponible, podrà fer-se càrrec de la petició; en cas contrari el client haurà d'esperar.
    - Una vegada s'aconsegueix trobar un treballador que processe aquesta petició, el client es mantindrà a l'espera de rebre el resultat (un altre missatge), moment en el qual podrà continuar amb la resta de la seua activitat.
- D'altra banda, els **treballadors** són altres processos que ofereixen serveis. Per a fer-ho construeixen (produeixen) un missatge per a anunciar la seua disponibilitat. Si hi ha algun client esperant, podran fer-se càrrec de la seua petició; en cas contrari quedaran a l'espera.
    - Una vegada el treballador aconsegueix una petició, la processarà[5] i retornarà un missatge amb dos significats: el resultat per al client i l'avís que el treballador torna a estar disponible.
    - Excepcionalment, el primer missatge enviat pel treballador únicament anunciarà la seua disponibilitat ja que no hi ha resultat que comunicar.
- És molt interessant que, per a fer d'intermediari entre aquests dos tipus de processos, apareix un tercer agent (proxy o ***broker***) que casa les peticions i oferiments de servei.

[4] Usem indistintament els termes proxy, intermediari i broker en aquesta pràctica
[5] L'activitat concreta a realitzar no és rellevant

A més, ha de vetlar per la correcció del sistema (cada resposta ha d'arribar al client concret que va formular la petició). Com a efecte addicional independitza uns processos d'uns altres.

2.2 Detallant i implementant l'esquema en NodeJS i ØMQ

De moment no detallarem les raons de disseny que ens porten a triar aquest esquema concret, però existeixen consideracions importants que donen valor a aquesta elecció.

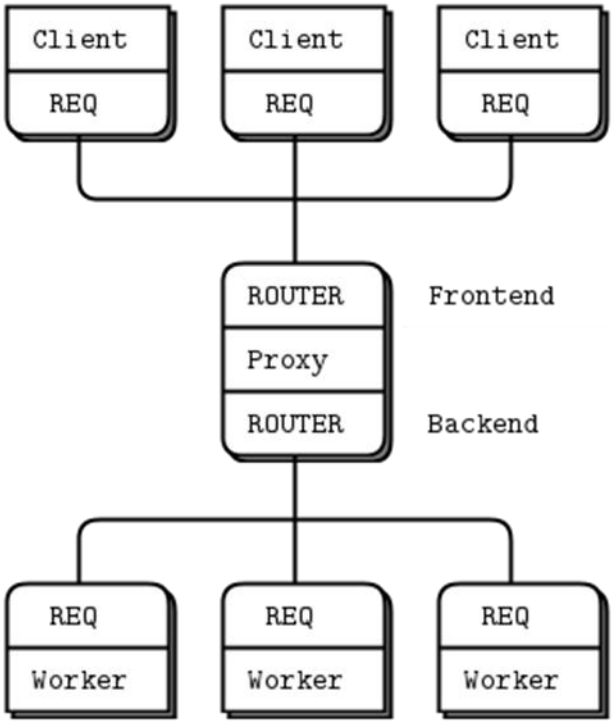

Figura 3: Esquema d'utilització d'un programa intermediari (*broker*) entre clients i servidors implementat amb el patró ROUTER-ROUTER.

En termes de comunicacions basades en **ØMQ,** aquest proxy, o intermediari, tindrà la següent funcionalitat:

- Presentarà un **socket ØMQ** de tipus **ROUTER**, on acceptarà connexions remotes dels clients (els qui li enviaran missatges de petició de servei). Anomenem ***frontend*** a aquest socket.
- Presentarà un altre **socket ØMQ** de tipus **ROUTER**, on acceptarà connexions remotes dels servidors o treballadors (els qui atendran les peticions de servei que els arriben). Anomenem ***backend*** a aquest socket.
- Quan un client, des del **seu socket ØMQ** de tipus **REQ**, envie una petició de servei (missatge) cap al socket *frontend*, el proxy seleccionarà un treballador, i li passarà el missatge pel socket ***backend***.
- Una vegada s'atenga la petició de servei (el missatge), el treballador que l'haja atès ho comunica, des del **seu socket ØMQ** de tipus **REQ**, mitjançant una petició enviada al proxy, pel socket ***backend***.
- Quan arribe aquesta petició d'un treballador pel socket ***backend***, el proxy l'ha de reexpedir pel socket ***frontend*** al client que va enviar la petició original.

- La gestió dels missatges descrita exigeix que el proxy conega les identitats de clients i treballadors.

Funcionament

Quant a sincronització condicional:

- Cada vegada que es registre un treballador nou, o un dels ja registrats responga la seua sol·licitud actual, el broker comprova si hi ha algun **client esperant**.
  a) En aquest cas, l'elimina de la cua i redirigeix la seua sol·licitud a aquest treballador lliure.
  b) En cas contrari, el treballador queda en la cua de treballadors disponibles.
- Cada vegada que arribe una sol·licitud d'algun client, el broker comprova si hi ha algun **treballador disponible**.
  a) En aquest cas, selecciona el primer treballador disponible i li envia la sol·licitud.
  b) En cas contrari, el client és situat en la cua de clients en espera.

Quant al temps de vida dels agents:

- Un client acabarà la seua execució quan reba resposta a totes les seues peticions.
- No obstant això, els servidors i el *broker* seguiran a l'espera de peticions de nous clients.

2.2.1 Descripció del client

El programa ***myclient.js*** contindrà el codi necessari per a executar un client (connectat al *frontend* del proxy).

Funcionalitat

- El client usarà un socket **REQ** per a enviar el seu missatge de petició de servei. Només enviarà una petició.
- El client quedarà a l'espera de rebre la resposta a la seua petició. Quan aquesta arribe, el client acabarà la seua execució.

2.2.1.1 Codi bàsic del client (myclient.js)

```
01:  // myclient in NodeJS
02:  var zmq       = require('zmq')
03:    , requester = zmq.socket('req');
04:
05:  var brokerURL = 'tcp://localhost:8059';
06:  var myID = 'NONE';
07:  var myMsg = 'Hello';
08:
09:  if (myID != 'NONE')
10:    requester.identity = myID;
11:  requester.connect(brokerURL);
12:  console.log('Client (%s) connected to %s', myID, brokerURL)
13:
14:  requester.on('message', function(msg) {
15:    console.log('Client (%s) has received reply "%s"', myID, msg.toString());
16:    process.exit(0);
17:  });
18:  requester.send(myMsg);
```

2.2.2 Descripció del treballador (*worker*)

El programa ***myworker.js*** contindrà el codi necessari per a executar un treballador (connectat al *backend* del *broker*).

Funcionalitat

- El treballador usarà un socket **REQ** per a comunicar-se amb *el broker*:
    a. Només enviarà una petició per a donar-se d'alta en *el broker* (en iniciar-se, informant de la seua disponibilitat).
    b. Enviarà, per cada petició de servei que reba, i una vegada aquesta siga atesa, una petició al *broker* perquè notifique al client corresponent que el servei ha sigut atès.
- El treballador quedarà en execució indefinida: sempre a l'espera de rebre peticions de servei.

2.2.2.1 Codi bàsic del worker (myworker.js)

```
01:  // myworker server in NodeJS
02:  var zmq = require('zmq')
03:    , responder = zmq.socket('req');
04:
05:  var backendURL = 'tcp://localhost:8060';
06:  var myID = 'NONE';
07:  var connText = 'id';
08:  var replyText = 'World';
09:
10:  if (myID != 'NONE')
11:    responder.identity = myID;
12:  responder.connect(backendURL);
13:  responder.on('message', function(client, delimiter, msg) {
14:      setTimeout(function() {
15:      responder.send([client,'',replyText]);
16:      }, 1000);
17:  });
18:  responder.send(connText);
```

2.2.3 Descripció del *broker* (proxy) i dels missatges

El programa ***mybroker.js*** contindrà el codi necessari per a executar el proxy o intermediari. La seua funcionalitat ja s'ha descrit a l'inici d'aquest apartat, però és moment de presentar algunes característiques més:

Funcionament

- Com és obvi, usarà 2 sockets **ROUTER** per a gestionar les comunicacions.
- Emmagatzemarà en alguna estructura de dades interna (cua, llista, taula…) els identificadors dels treballadors que li hagen enviat missatges de disponibilitat.
- Quan reba peticions de servei (en *el seu frontend*), verificarà si disposa de treballadors que puguen atendre-ho i, en tal cas, seleccionarà al més adequat (en termes d'equilibri

de càrrega), reexpedint-li la petició (a través del *seu backend*). En cas contrari, guardarà la petició en una cua.

Prèviament s'ha de crear un directori ***zmqavanzado***, al que s'afegiran els programes necessaris per a posar en funcionament l'esquema descrit. Aquests programes, que es mostren a continuació **de forma abreujada**[6], es diuen: ***myworker.js***, ***mybroker.js*** i ***myclient.js***.

2.2.3.1 Codi bàsic del broker (mybroker.js)

```
01:  // ROUTER-ROUTER request-reply broker in NodeJS
02:  var zmq      = require('zmq')
03:    , frontend = zmq.socket('router')
04:    , backend  = zmq.socket('router');
05:
06:  var fePortNbr = 8059;
07:  var bePortNbr = 8060;
08:  var workers = [];
09:  var clients = [];
10:
11:  frontend.bindSync('tcp://*:'+fePortNbr);
12:  backend.bindSync('tcp://*:'+bePortNbr);
13:
14:  frontend.on('message', function() {
15:    var args = Array.apply(null, arguments);
16:    if (workers.length > 0) {
17:      var myWorker = workers.shift();
18:      var m = [myWorker,''].concat(args);
19:      backend.send(m);
20:    } else
21:      clients.push( {id: args[0],msg: args.slice(2)});
22:  });
23:
24:  function processPendingClient(workerID) {
25:    if (clients.length>0) {
26:      var nextClient = clients.shift();
27:      var m = [workerID,'',nextclient.id,''].concat(nextClient.msg);
28:      backend.send(m);
29:      return true;
30:    } else
31:      return false;
32:  }
33:
34:  backend.on('message', function() {
35:    var args = Array.apply(null, arguments);
36:    if (args.length == 3) {
37:      if (!processPendingClient(args[0]))
38:        workers.push(args[0]);
39:    } else {
40:        var workerID = args[0];
41:        args = args.slice(2);
42:        frontend.send(args);
43:        if (!processPendingClient(workerID))
44:          workers.push(workerID);
45:    }
46:  });
```

[6] En un annex pots trobar una versió anotada amb comentaris

2.3 Funcionament del *broker* amb el patró ROUTER - ROUTER

Seguidament repassem el funcionament dels patrons **REQ-ROUTER** i **ROUTER-REQ**, ja explicats en el seminari de ØMQ, amb la finalitat de comprendre com s'ha de fer arribar la petició del client al servidor, així com la resposta del servidor al client.

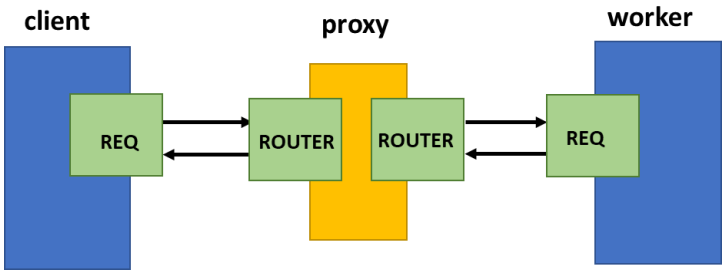

Figura 4: Esquema global dels agents del patró ROUTER-ROUTER.

Vegem primer el camí d'anada de la petició del client:

2.3.1 Enviament de la petició del client

Suposem que el programa client (*myclient.js*) envia el missatge de petició (**"Work"**) des del seu socket REQ al socket ROUTER del *broker* (*mybroker.js*). Suposarem que el client té com a identitat **"CLIENT1".**

NOTA: per a especificar aquesta identitat, els clients han d'executar la següent instrucció abans d'establir la connexió amb el socket ROUTER del *broker* (línia 10 de *myclient.js*):

```
…
requester.identity=myID; // suposem "CLIENT1"
…
requester.connect(brokerURL);
…
```

Tal com es mostra en *la figura 5*, el socket REQ afig un segment amb un delimitador buit al contingut del missatge enviat pel client. Per la seua banda, el socket ROUTER rep els dos segments en la cua de recepció associada a "CLIENT1". Seguidament, el socket ROUTER afig un nou segment, al missatge rebut, amb la identitat del client. D'aquesta forma *mybroker.js* rebrà un missatge amb tres segments [*"CLIENT1"," ", "Work"*], que queda emmagatzemat en la variable ***args*.**

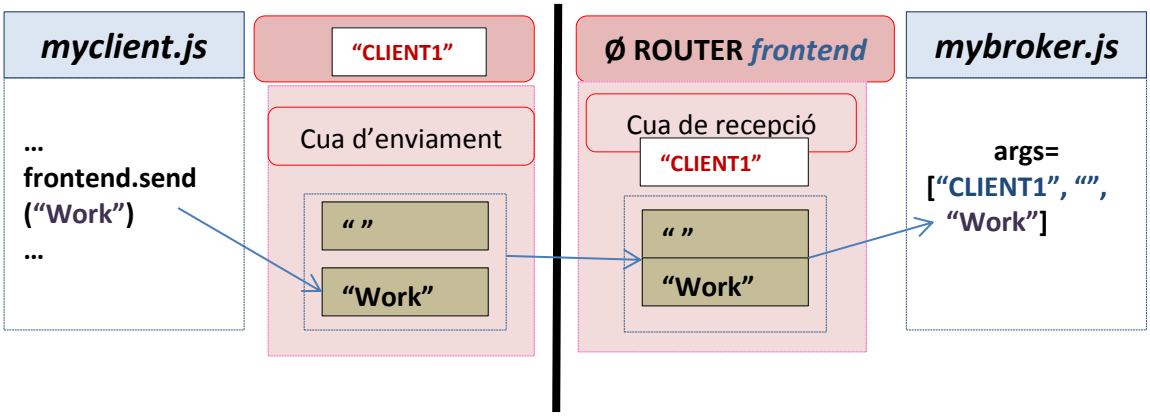

Figura 5: Enviament de la petició del client (*myclient.js*) al *broker* (*mybroker.js*) utilitzant el patró REQ-ROUTER.

Vegem ara **com es transmet la petició del *broker* al servidor**:

com el socket backend del *broker* és de tipus ROUTER, *mybroker.js* serà capaç de triar a quin treballador envia la petició. Per a fer això haurà de proporcionar la identitat del treballador. Aquesta identitat més un delimitador buit han de ser inserits com a primers segments del nou missatge (vegeu el contingut en roig de la variable **args** de *mybroker.js* en *la figura 6*).

NOTA: En aquest cas la identitat del treballador no és introduïda automàticament pel socket ØMQ, sinó que ha de ser seleccionada explícitament per *mybroker.js,* d'entre la llista de treballadors. Aquesta identitat és necessària perquè el socket ROUTER puga col·locar el missatge en la cua d'enviament associada al treballador. Lògicament, el treballador seleccionat passarà a no disponible, és a dir, no podrà atendre noves peticions mentre no arribe la seua resposta.

Si suposem que la identitat del treballador seleccionat és “*WORKER1*”, el missatge que el *broker* ha d'enviar a la cua d'enviament del backend serà, [*“WORKER1”,””, “CLIENT1”,””, “Work”*] (vegeu *figura 6* a continuació).

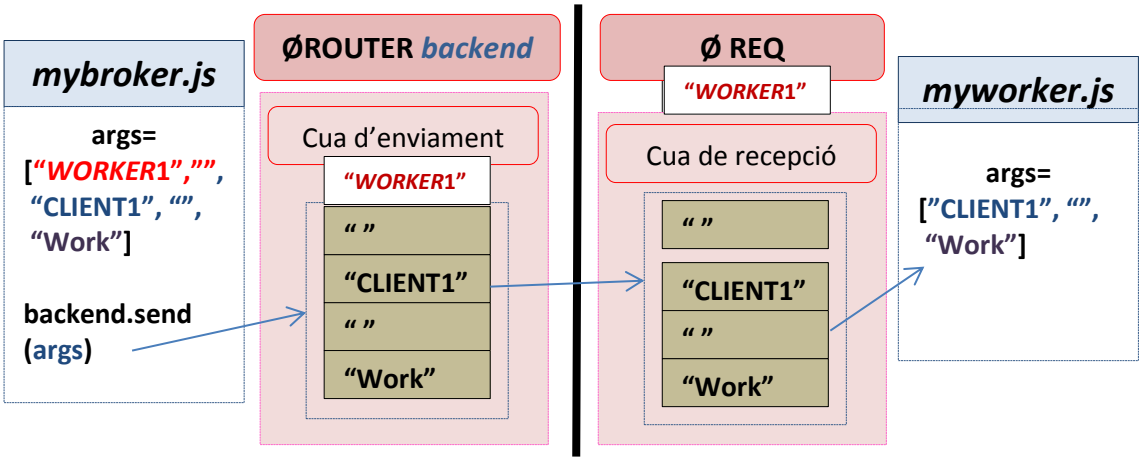

Figura 6: Enviament de la petició del client des del *broker* (*mybroker.js*) al servidor (*myworker.js*) utilitzant el patró ROUTER-REQ.

Tal com es mostra en *la figura 6*, el socket ROUTER transmetrà el missatge utilitzant la cua associada al "*WORKER1*" i envia el missatge al socket REQ del treballador sense el primer segment (és a dir, només els quatre últims segments). Aquest descompon el missatge de manera que el programa *myworker.js* no rep el primer segment, corresponent al primer delimitador buit. Aquest segment és guardat pel socket REQ per a ser enviat de tornada juntament amb la contestació a la petició. Per tant, el treballador rep un missatge amb tres segments, [*"CLIENT1"," ", "Work"*].

2.3.2 Enviament de la resposta del servidor

Tal com es mostra en *la figura 7,* el treballador haurà de contestar amb un missatge similar, de tres segments, canviant el segment amb el contingut de la petició pel contingut de la resposta (suposarem que aquesta resposta és la cadena "Done"). Per tant, el treballador enviarà com a resposta el missatge [*"CLIENT1"," ", "Done"*] i a la cua d'enviament del socket REQ arriba la resposta del treballador amb aquests tres segments, al que se li afig el delimitador buit. La cua de recepció del socket ROUTER del backend afig la identitat del treballador pel *que mybroker.js* rep un missatge amb cinc segments ([*"WORKER1", "", "CLIENT1"," ", "Done"*]), que queda emmagatzemat novament en la variable ***args*.**

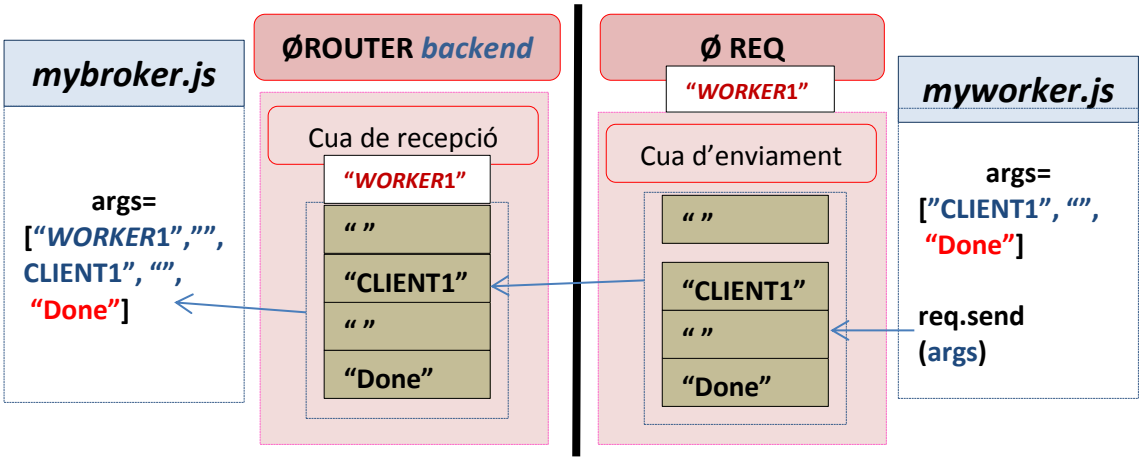

Figura 7: Enviament de la resposta del servidor (*myworker.js*) al *broker* (*mybroker.js*) utilitzant el patró ROUTER-REQ.

L'últim pas és l'enviament de la resposta al client. Perquè aquesta arribe correctament és necessari que el primer segment del missatge continga la identitat d'aquest client. És per això que el *broker* ha d'extraure els dos primers segments d'args per a enviar la resposta al client.

NOTA: La informació extreta (concretament la identitat del treballador) serà utilitzada per a col·locar al treballador que ha atès la petició en estat disponible novament.

Finalment, el nou contingut d'args és enviat a la cua d'enviament del socket ROUTER del *broker* corresponent al "*CLIENT1*" (tal com es mostra en *la figura 8*), per la qual cosa aquest primer segment és extret. Això vol dir que a la cua de recepció del socket REQ del client arriben només dos segments, [*"", "Done"*]. Finalment, el socket REQ extrau el delimitador buit i el programa client (*myclient.js*) solament rep la resposta del servidor ("*Done*").

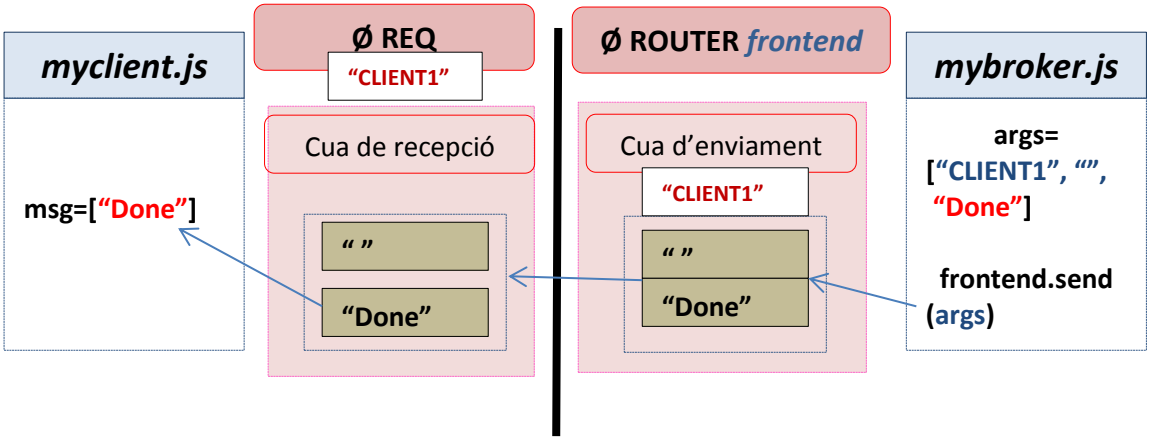

Figura 8: Enviament de la resposta del servidor des del *broker* (*mybroker.js*) al client (*myclient.js*) utilitzant el patró REQ-ROUTER.

La següent *figura 9* mostra, finalment, un esquema conjunt de l'enviament de la petició i la recepció de la resposta:

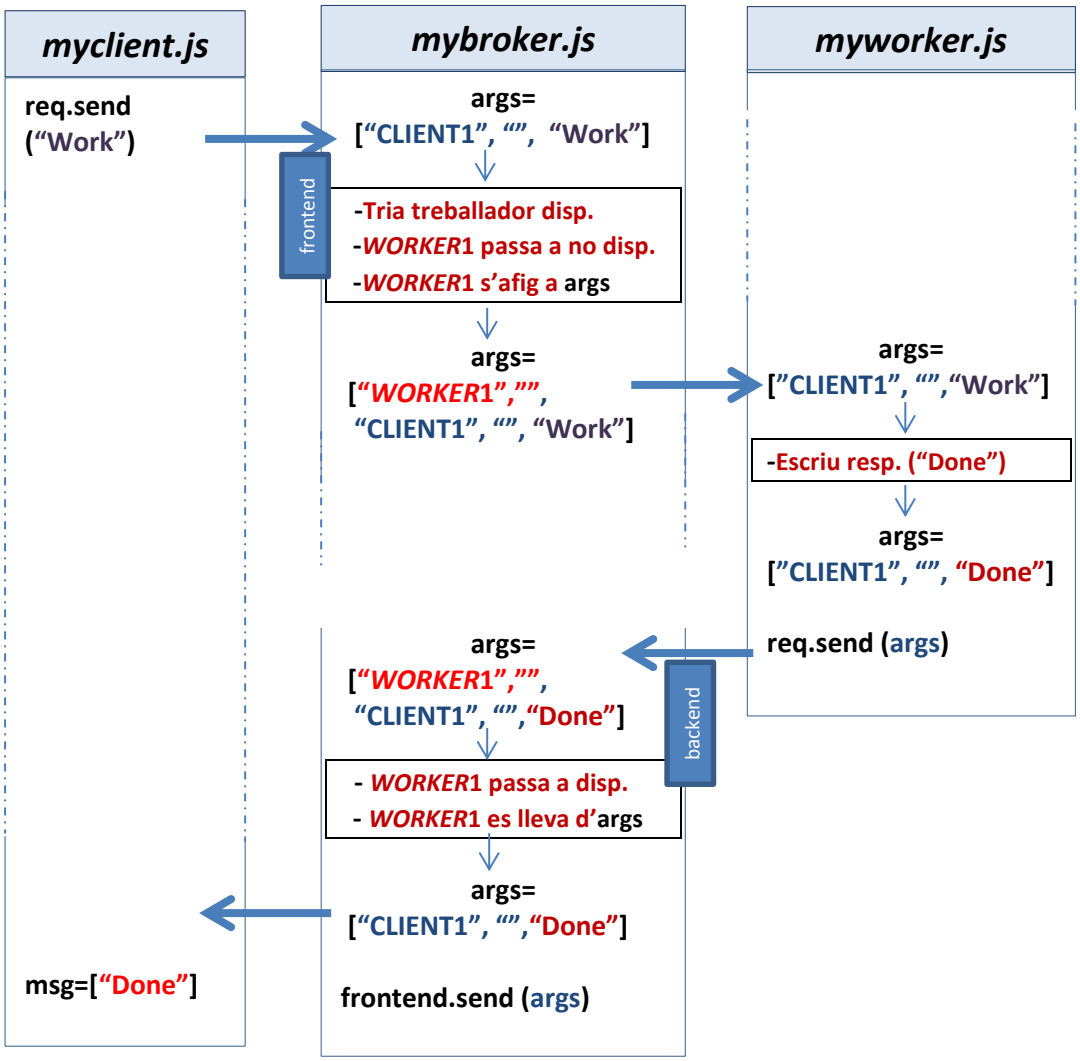

Figura 9: *Esquema d'enviament i recepció de resposta per a l'arquitectura mostrada en la figura 4.*

2.4 Implementació i proves preliminars

La finalitat d'aquestes primeres proves és comprovar que els missatges es transmeten adequadament entre client i treballador, a través del *broker*, així com que es respecta allò relatiu a transformacions del missatge a través dels sockets ROUTER, tal com s'ha explicat.

Per a provar el funcionament de les nostres primeres versions dels programes *myworker.js*, *mybroker.js* i *myclient.js*, seguirem una estratègia prudent i conservadora: es posarà en execució el proxy, i únicament un treballador i un client.

Desgraciadament el codi de partida no ens ofereix ajuda per a conèixer què està ocorrent en el seu interior quan ho posem en funcionament. Si hem fet alguna cosa malament, haurem de

repassar les instruccions i col·locar sentències que ens informen de l'estat intern (variables, missatges) a cada moment. Això fa necessari el mode *verbose*.

2.4.1 Mode *verbose*

Aquest mode permetrà obtenir alguna cosa similar a una traça de l'execució dels programes conforme es processa una petició de servei. En l'Annex 2 es mostra un exemple de l'eixida per consola de l'execució dels programes en *mode verbose*. Aquest exemple es presenta com a referència: no perquè la vostra implementació genere exactament aqueixa eixida.

Una possibilitat d'implementació d'aquest mode aplicat a treballador i broker podria ser:

Per a **myworker.js**

```
if (verbose)
  console.log('worker (%s) connected to %s', myID, backendURL);

responder.on('message', function(client, delimiter, msg) {
  if (verbose) {
    console.log('worker (%s) has received request "%s" from client (%s)',
      myID, msg.toString(), client);
  }
  setTimeout(function() {
    responder.send([client,'',replyText]);
    if (verbose)
      console.log('worker (%s) has sent its reply "%s"',
        myID, replyText);
  }, 1000);
});
responder.send(connText);
if (verbose)
  console.log('worker (%s) has sent its first connection message: "%s"',
    myID, connText);
```

I per a **mybroker.js** col·loquem sentències per a observar el contingut dels missatges cada vegada que arriben o s'envien. Per a facilitar la visualització del seu contingut, emprem la funció **showMessage()** d'auxfunctions1718 **.js**[7]

```
function showMessage(msg) {
  msg.forEach( (value,index) => {
    console.log( '     Segment %d: %s', index, value );
  })
}
```

Invocant-se, p. ex., d'aquesta manera:

```
var aux = require("./auxfunctions1718.js");
…
function sendToWorker(msg) {
  if (verbose) {
    console.log('Sending client (%s) request to worker (%s) through backend.',
      msg[2], msg[0]);
    aux.showMessage(msg);
  }
  backend.send(msg);
}
```

[7] Serà necessària una sentència `require` al començament del programa

2.4.2 Parametritzant el codi

Com sabem, moltes aplicacions poden prendre arguments des de la línia d'ordres. Podem parametritzar els programes perquè deixen de ser constants les informacions relacionades amb ports de connexió, identificadors i uns altres. De fet, podem emprar una manera *híbrida*, que aprofita l'avaluació lògica amb curtcircuit en Javascript, en el qual una expressió tal com A ||B s'avalua a A si aquest té valor, o B si no. Així:

```
var args = process.argv.slice(2);
var fePortNbr = args[0] || 8059;
```

S'interpreta com: col·locar els paràmetres d'invocació en `args`. `fePortNbr` pren com a valor el primer paràmetre, però si falta aqueix paràmetre llavors s'inicialitza a `8059`.

Els noms que tindran els fitxers resultants d'aquestes dues darreres extensions (mode *verbose* i parametrització) seran: myclient_vp.js, mybroker_vp.js i myworker_vp.js. A continuació es mostra l'especificació dels paràmetres de cada component juntament amb el codi que permet incorporar-ho a cadascun:

Arguments i codi de myclient_vp.js

a. URL de l'endpoint del ROUTER *frontend* en *el broker* (proxy).
b. Cadena de caràcters (*string*) que represente la identitat del client.

> **NOTA:** Recordem que el client està connectat al *broker* a través del seu socket ROUTER, el qual assigna identitats a cadascun dels sockets connectats al *seu endpoint*. En passar aquest paràmetre triem la cadena de text que constitueix la identitat del client, en comptes de deixar que siga el socket ROUTER qui l'establisca per omissió.

c. Text de la petició de servei, per exemple: '**WORK**'.

```
04:  var args = process.argv.slice(2);
05:  var brokerURL = args[0] || 'tcp://localhost:8059';
06:  var myID = args[1] || 'NONE';
07:  var myMsg = args[2] || 'Hello';
```

Arguments i codi de myworker_vp.js

a. URL de l'endpoint del ROUTER *backend* en *el broker* (proxy).
b. Cadena de caràcters (*string*) que represente la identitat del treballador.
c. Text d'un missatge de disponibilitat[8], per exemple: '***READY***'.
d. Text d'un missatge d'atenció de servei[9], per exemple: '***DONE***'.

```
04:  var args = process.argv.slice(2);
05:  var backendURL = args[0] || 'tcp://localhost:8060';
06:  var myID = args[1] || 'NONE';
07:  var connText = args[2] || 'id';
08:  var replyText = args[3] || 'World';
```

Arguments i codi de mybroker_vp.js

[8] Missatge de petició de donar-se d'alta en *el broker*.
[9] Missatge de petició al *broker* per a notificar a un client que se li ha donat servei.

a. Port de comunicació amb els clients (*frontend*).
b. Port de comunicació amb els treballadors (*backend*),

```
05:   var args = process.argv.slice(2);
06:   var fePortNbr = args[0] || 8059;
07:   var bePortNbr = args[1] || 8060;
```

Proves

Cal comprovar el funcionament del patró ROUTER-ROUTER, executant els programes en una mateixa màquina (la teua virtual) o en dues màquines diferents (el broker haurà d'executar-se en la virtual). Almenys, s'ha de verificar:

a) El funcionament amb un client i dos servidors. Per a fer això, indicar en executar els servidors, en el seu últim argument (resposta a enviar): "***DONE1***", "***DONE2***", respectivament. Comproveu el repartiment de treball en l'atenció de les peticions del client.
b) El funcionament amb dos clients i un servidor. Per a fer això, indicar en executar els clients, en el seu últim argument (petició a enviar): "***WORK1***", "***WORK2***", respectivament. Comproveu si el servidor atén equitativament les peticions dels clients.
c) El funcionament amb dos clients i tres servidors, indicant també missatges identificatius tant de clients com de servidors. Comproveu si queden peticions no ateses, així com el repartiment de treball entre els servidors.

2.5 Més implementació i proves completes

Si s'ha aconseguit superar satisfactòriament les proves del punt anterior, estem en condicions d'incrementar el nombre de treballadors i de clients. Podrem llavors verificar si el nostre *broker* reparteix equitativament el treball entre els integrants del *seu pool* de treballadors.

Es recomana fer les primeres proves amb un nombre moderat de treballadors i clients. Per exemple, amb 3 treballadors i 10 clients. Seria convenient modificar el *broker* de manera que porte el compte del nombre de peticions ateses per cada treballador, i que mostre en finalitzar (per consola) una taula amb els identificadors dels treballadors i les peticions ateses per cadascun.

Des d'una mateixa terminal, podem llançar[10] fàcilment l'execució simultània de, per exemple, dos treballadors mitjançant:

```
bash-4.1$ node myworker_vp tcp://la-meua-IP-externa:8060 WORKER1 Ready DONE &
          node myworker_vp tcp://la-meua-IP-externa:8060 WORKER2 Ready OK &
```

No obstant això, cal anar amb compte amb l'execució *batch*, o en segon pla, de programes que no tenen una conclusió explícita. Seria recomanable modificar el treballador (myworker.js) de

[10] Suposem una IP com 192.168.1.1, a la qual, en general denominem `la-meua-IP-externa` per a diferenciar-la de localhost

manera que tinguera una "vida limitada" (fixant un temporitzador amb la funció *setTimeout,* que llance el tancament del procés: *process.exit();*).

Si es vol fer una prova amb un nombre significativament gran de clients, per exemple, 100 clients, s'ha de recórrer a una altra estratègia d'execució simultània dels programes: implementar un ***shell script*** [11].

Considerem el següent *script* senzill:

```
01:    #!/bin/bash
02:
03:    number=1
04:    while [ $number -lt $1 ]; do
05:        echo "ZeroMQ != $number MQ"
06:        number=$((number + 1))
07:    done
08:    echo "ZeroMQ == 0MQ"
```

A partir de l'esquema d'aquest *script*, no hauria de ser difícil implementar un parell de scripts, anomenats ***myclients_script.sh*** i ***myworkers_script.sh***, que ens permeten, mitjançant una sola ordre (la de la seua invocació), executar un nombre bastant major de clients i treballadors, respectivament.

2.6 Modificacions de curt abast

2.6.1 Estadístiques

Serà necessari acumular la informació necessària per a, quan se sol·licite, poder calcular els resultats. Per a la **primera** part, afegim en el broker:

```
// Array with counters of how many requests have been processed
// by each worker.
var requestsPerWorker = [];
```

En `backend.on`, quan es tracta de l'anunci per part d'un nou treballador (`args.length == 3`), anem a inicialitzar `requestsPerWorker[args[0]]=0`; però en cas contrari incrementem el comptador amb `requestsPerWorker[args[0]]++`

Per a la segona part (**calcular i comunicar**, p.ex. en finalitzar amb CTRL-C), afegim al broker:

```
// Function that shows the service statistics.

function showStatistics() {
      var totalAmount = 0;
      console.log('Current amount of requests served by each worker:');
      for (var i in requestsPerWorker) {
            console.log('   %s : %d requests', i, requestsPerWorker[i]);
            totalAmount += requestsPerWorker[i];
      }
      console.log('Requests already served (total): %d', totalAmount);
}
```

[11] Una bona referència sobre aquest tema és: http://linuxcommand.org/writing_shell_scripts.php. On es pot trobar el relatiu a bucles: http://linuxcommand.org/wss0120.php#loops

```
// Show the statistics each time [ctrl]+[C] is pressed.
process.on('SIGINT', showStatistics);
```

2.6.2 Encadenament de brokers

Es desitja realitzar una modificació que substitueix el broker per dos components (broker1 i broker2) encadenats.

- Els clients envien peticions a broker1, aquest li les passa a broker2 i, finalment, broker2 li les fa arribar als treballadors.
- El camí de tornada és exactament l'invers.

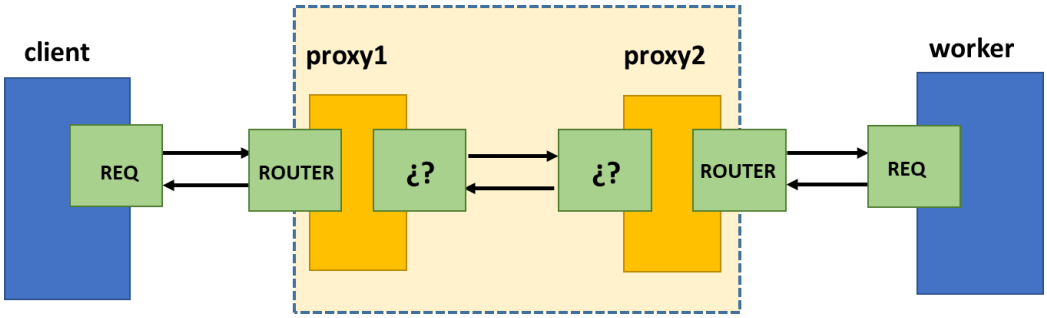

Figura 10: Esquema global dels agents del patró ROUTER-ROUTER, amb brokers encadenats, sense selecció del patró intern de comunicació entre brokers.

Es demana implementar el codi de tots dos brokers de manera que el conjunt es puga comunicar amb les mateixes característiques externes que ja observaven clients i treballadors, especialment quant a sincronització. S'insisteix a triar acuradament el model de comunicació intern dels brokers, perquè no hauria de modificar-se la percepció externa del seu comportament.

A més haurà d'activar-se un mode *verbose* en tots dos intermediaris perquè, almenys:

- Per a cada missatge de petició dels clients que arriba a broker1, aquest ha d'escriure un missatge en pantalla que identifique a aquest client (*el client XX ha realitzat una petició*)
- Per a cada missatge de resposta procedent dels workers, broker1 ha de mostrar un altre missatge similar (*al client XX se li va a retornar una resposta*)

3 NOVES OPORTUNITATS DES DE L'ESQUEMA ROUTER-ROUTER

Es pretén descriure nous objectius a cobrir prenent com a punt de partida el sistema descrit anteriorment. El dos primers es presenten (i resolen) de forma exhaustiva, mentre que el tercer es presenta a un nivell que permeta entendre i aplicar les propostes de solució que es donen.

Els problemes a resoldre des de l'esquema ROUTER-ROUTER són:

1. **Tipus de treball**. Quan les peticions referencien habilitats específiques, solament alguns treballadors hauran manifestat la seua capacitat per a realitzar-les.
2. **Batec** *(heartbeat)*. En un grup dinàmic de treballadors, existeix la possibilitat que algun deixe de contestar.
3. **Equilibrat de càrrega** *(load balancing)*. Conèixer el grau de saturació dels treballadors ens permet seleccionar el més adequat per al treball a assignar.

3.1 Tipus de treball

En alguns escenaris no tots els agents disposen dels mateixos recursos i capacitats. Potser trobem nodes que compten amb el suport d'una targeta gràfica amb milers de processadors, uns altres amb grans recursos d'emmagatzematge, o alguns que es diferencien entre si per altres criteris.

- Com a cas extrem, el broker podria analitzar cada petició i deduir a quin treballador especialitzat enviar-la. No s'indica com pot realitzar-se aquesta anàlisi, però ha de ser alguna cosa senzilla per a no penalitzar el funcionament global. Tampoc s'especifica com s'esbrina l'especialització de cada treballador.
- Com a cas pràctic, els clients poden especificar el tipus de petició en cada sol·licitud, i els treballadors poden indicar quins tipus de peticions poden atendre. Aquesta explicitació obliga a senzilles modificacions en els missatges de tots dos extrems.

Ens centrarem en aquesta última aproximació perquè és més concreta. Reduïm els tipus a 3 (per exemple {**R, G, B**}) i ho afegirem als missatges de petició dels clients i de disponibilitat dels treballadors.

Els dos tipus de components (clients i treballadors) reben un tipus `classID` en la seua invocació com a últim argument.

- Els **clients** especifiquen el `classID` com un valor que forma part del missatge.
- Anàlogament, els **treballadors** anuncien el tipus que poden processar en el missatge de registre. Per exemple, per a informar que poden processar peticions de tipus B faran `s.send('B')`.

Desenvoluparem una nova versió (`myclient1`, `mybroker1` i `myworker1`) de cada component segons el comportament descrit, sense necessitat de sockets addicionals.

Un exemple d'invocació de `myworker1.js` és:

```
node myworker1.js Worker1 localhost:8099 R
```

que, en aquest exemple, hauria d'interpretar-se com:

- L'identificador d'aquest worker és `Worker1`
- Connecta amb el broker en `tcp://localhost:8099`
- Admet processar peticions de tipus `R`

En el cas dels clients (`myclient1.js`), la seua invocació podria ser:

```
node myclient1.js Client1 localhost:8098 G
```

que, en aquest exemple, hauria d'interpretar-se com:

- L'identificador d'aquest client és `Client1`
- Connecta amb el broker en `tcp://localhost:8098`
- Anuncia que la petició de servei és del tipus `G`

Per a comprovar aquesta part és convenient arribar a executar concurrentment una instància de mybroker1, i diverses[12] de clients i treballadors com...

```
node mybroker1.js 8098 8099
node myworker1.js Worker1 localhost:8099 R
node myworker1.js Worker2 localhost:8099 G
node myclient1.js Client1 localhost:8098 G
node myclient1.js Client2 localhost:8098 G
node myclient1.js Client3 localhost:8098 R
node myclient1.js Client4 localhost:8098 G
```

3.1.1 Punts clau de la solució

No hi ha molta diferència entre disposar de N tipus de treball i disposar de N brokers. Els clients i treballadors interessats en el tipus de treball K contactarien amb el broker K per a poder interactuar.

Suposant que aquesta solució no ens satisfaça per qüestions que, de moment, no són especialment interessants, el plantejament amb un sol broker pot emular el comportament de diversos simplement disposant de cues diferents (`workers[]` i `clients[]`) per a tipus de treball diferents.

3.1.2 Solució

Per a ser consistent amb un plantejament centrat en el codi representatiu, com en el codi bàsic dels tres agents, prescindim de l'argument per a especificar el tipus (suposem '`B`') i l'incloem com una constant en el codi de clients i treballadors.

Les modificacions sobre els dos agents són mínimes. Respecte al broker, pràcticament és suficient amb substituir les cues per vectors de cues, seleccionant el vector concret mitjançant l'últim segment (`classID`) del missatge que es reba.

[12] És interessant que cadascuna s'execute en una finestra diferent per a no interferir

3.1.2.1 myclient1.js

```
01:  // myclient1 in NodeJS , classID='B'
02:  var zmq       = require('zmq')
03:    , requester = zmq.socket('req');
04:
05:  var brokerURL = 'tcp://localhost:8061';
06:  var myID = 'NONE';
07:  var myMsg = 'Hello';
08:  var classID = 'B';
09:
10:  if (myID != 'NONE')
11:    requester.identity = myID;
12:  requester.connect(brokerURL);
13:  console.log('Client (%s), class (%s) connected to %s', myID, classID,
     brokerURL)
14:
15:  requester.on('message', function(msg) {
16:    console.log('Client (%s) has received reply "%s"', myID, msg.toString());
17:    process.exit(0);
18:  });
19:  requester.send([myMsg, classID]);
```

3.1.2.2 myworker1.js

```
01:  // myworker1 server in NodeJS , classID='B'
02:  var zmq = require('zmq')
03:    , responder = zmq.socket('req');
04:
05:  var backendURL = 'tcp://localhost:8062';
06:  var myID = 'NONE';
07:  var connText = 'id';
08:  var replyText = 'World';
09:  var classID = 'B';
10:
11:  if (myID != 'NONE')
12:    responder.identity = myID;
13:  responder.connect(backendURL);
14:  responder.on('message', function(client, delimiter, msg) {
15:      setTimeout(function() {
16:      responder.send([client,'',replyText, classID]);
17:      }, 1000);
18:  });
19:  responder.send([connText, classID]);
```

3.1.2.3 mybroker1.js

```
01:  // ROUTER-ROUTER request-reply broker in NodeJS with classIDs
02:  var zmq      = require('zmq')
03:    , frontend = zmq.socket('router')
04:    , backend  = zmq.socket('router');
05:
06:  var fePortNbr = 8061;
07:  var bePortNbr = 8062;
08:  var workers = [];
09:  var clients = [];
10:  const classIDs = ['R', 'G', 'B']
11:  for (var i in classIDs){
12:     workers[classIDs[i]]=[];
13:     clients[classIDs[i]]=[];
14:  }
15:
16:  frontend.bindSync('tcp://*:'+fePortNbr);
17:  backend.bindSync('tcp://*:'+bePortNbr);
18:
19:  frontend.on('message', function() {
20:    var args = Array.apply(null, arguments);
21:    var classID = args.pop();
```

```
22:     if (workers[classID].length > 0) {
23:       var myworker = workers[classID].shift();
24:       var m = [myworker,''].concat(args);
25:       backend.send(m);
26:     } else
27:       clients[classID].push( {id: args[0],msg: args.slice(2)});
28:   });
29:
30:   function processPendingClient(workerID, classID) {
31:     if (clients[classID].length>0) {
32:       var nextClient = clients[classID].shift();
33:       var m = [workerID,'',nextclient.id,''].concat(nextClient.msg);
34:       backend.send(m);
35:       return true;
36:     } else
37:       return false;
38:   }
39:
40:   backend.on('message', function() {
41:     var args = Array.apply(null, arguments);
42:     var classID = args.pop();
43:     if (args.length == 3) {
44:       if (!processPendingClient(args[0], classID))
45:         workers[classID].push(args[0]);
46:     } else {
47:         var workerID = args[0];
48:         args = args.slice(2);
49:         frontend.send(args);
50:         if (!processPendingClient(workerID, classID))
51:           workers[classID].push(workerID);
52:     }
53:   });
```

Hi ha algunes millores i ampliacions menors com la parametrització dels agents i la incorporació d'un mode *verbose*, però **el realment interessant** seria la realització de les següents modificacions (primer apartat) i la resposta a la pregunta que es planteja en el segon:

1. Si no es coneix per endavant el nombre i identificació dels tipus de treball, haurà de modificar-se el codi del broker per a adaptar-se a una d'aquestes 2 possibilitats:
   - Els tipus es donen a conèixer com a arguments en la invocació del broker. Així, una ordre com `node broker A B C D E F` hauria de provocar que el broker tinguera 6 cues, una per al tipus de treball A, una altra per als del tipus B, i així successivament.
   - Els tipus es van coneixent com a contingut dels missatges que arriben al broker, tant des dels clients com des dels treballadors: cada vegada que arriba un missatge, comprova si existeix una cua associada al seu tipus. Si no n'hi ha cap, la crea. Per simplicitat, mai es destrueix una cua encara que es buide.
2. Discuteix, sense implementar, les dificultats d'implementació que tindria la possibilitat que un **treballador** puga oferir-se per a processar indistintament treballs **de diversos tipus**.

3.2 Batec

L'objectiu d'implantar aquest tipus de mecanisme és que el *broker* puga detectar caigudes dels treballadors. D'aquesta forma es podran prevenir fallades i detectar les peticions que han quedat pendents de resposta. Es tracta d'incloure un mecanisme senzill de tolerància a fallades gestionat a través del *broker*.

Per a resoldre-ho, com a primera alternativa, es podria utilitzar un socket ROUTER auxiliar en *el broker* i un altre socket REQ auxiliar en els treballadors, exclusivament per a implementar aquest mecanisme. El *broker* enviaria **periòdicament** (*polling*) els missatges d'*heartbeat* als treballadors a través del seu ROUTER auxiliar, i aquests envien el missatge de resposta a través del seu socket REQ auxiliar.

- Si no es rep resposta durant aquest període, se suposarà que el treballador ha caigut, reaccionant en conseqüència.
- El codi dels treballadors ha d'incloure la resposta a aquests missatges.
- És una opció costosa però previsora (difícilment enviarà peticions a treballadors caiguts).

No obstant això, el mateix intercanvi de missatges que existeix com a activitat de fons pot interpretar-se com a constatació que els treballadors implicats estan *vius*: no és necessari interrogar a qui acaba de demostrar que es troba en funcionament.

- Algunes aproximacions a aquest problema solament envien missatges d'*heartbeat* als treballadors dels qui no es té notícia recent.
- No és molt més complex que l'anterior i pot estalviar missatges.

En aquesta línia, un cas especial consisteix a preguntar-se únicament per l'estat dels treballadors quan se'ls envia una petició, però tarden un temps excessiu a comunicar el resultat.

- És una opció lleugera que ens obliga a un pla de recuperació si un treballador cau, ja que cal reexpedir la sol·licitud a un altre.
- Únicament requereix modificacions en el codi del broker.
- És la que descriurem en detall per a incorporar-la al nostre codi.

3.2.1 Claus de la solució

En cada ocasió en què l'intermediari reenvia la sol·licitud d'un client a un treballador, ha de recordar aquesta correspondència i col·locar un *timeout* (`answerInterval`) màxim per a rebre la resposta. Si durant aquest interval no es rep res, es considerarà que el treballador ha fallat, i la petició es reenviarà a un altre treballador disponible.

3.2.2 Solució

En el llistat, basat en el codi original de l'intermediari, es destaquen les modificacions realitzades juntament amb una xicoteta descripció.

S'ha realitzat una reestructuració incorporant una funció `sendToWorker()`, que "recobreix" a `backend.send` per a anotar al treballador pendent de resposta i iniciar el temporitzador, i `sendRequest()`, que "recobreix" a `sendToWorker()` per a triar el treballador candidat.

```
01:  // mybroker2.js: ROUTER-ROUTER request-reply broker in NodeJS
02:  // Worker availability-aware variant.
03:  // It interacts with the other original agents: myclient.js and myworker.js
04:  var zmq      = require('zmq')
05:    , frontend = zmq.socket('router')
06:    , backend  = zmq.socket('router');
07:
08:  var fePortNbr = 8059;
09:  var bePortNbr = 8060;
10:
11:  var workers = [];
12:  var clients = [];
13:
14:  // Reply awaiting period, in ms.
15:  const answerInterval = 2000;
16:  // Array of busy workers. Each slot contains an array with all
17:  // the segments being needed for resending the current message
18:  // being processed by that worker.
19:  var busyWorkers = [];
20:
21:  // Send a message to a worker.
22:  function sendToWorker(msg) {
23:    var myWorker = msg[0];
24:    // Send the message.
25:    backend.send(msg);
26:    // Initialise busyWorkers slot object.
27:    busyWorkers[myWorker] = {};
28:    // Recall that such message has been sent.
29:    busyWorkers[myWorker].msg=msg.slice(2);
30:    // Set a timeout of its response.
31:    busyWorkers[myWorker].timeout=
32:      setTimeout(generateTimeoutHandler(myWorker),answerInterval);
33:  }
34:
35:  // Function that sends a message to a worker, or
36:  // holds the message if no worker is available now.
37:  // Parameter 'args' is an array of message segments.
38:  function sendRequest(args) {
39:    if (workers.length > 0) {
40:      var myWorker = workers.shift();
41:      var m = [myWorker,''].concat(args);
42:      sendToWorker(m);
43:    } else {
44:      clients.push( {id: args[0],msg: args.slice(2)});
45:    }
46:  }
47:
48:  // Function that creates the handler for a reply
49:  // timeout.
50:  function generateTimeoutHandler(workerID) {
51:    return function() {
52:      // Get the message to be resent.
53:      var msg = busyWorkers[workerID].msg;
54:      // Remove that slot from the busyWorkers array.
55:      delete busyWorkers[workerID];
56:      // Resend that message.
57:      sendRequest(msg);
58:    }
59:  }
60:
61:  frontend.bindSync('tcp://*:'+fePortNbr);
62:  backend.bindSync('tcp://*:'+bePortNbr);
63:
64:  frontend.on('message', function() {
65:    var args = Array.apply(null, arguments);
66:    sendRequest(args);
```

```
67:   });
68:
69:   function processPendingClient(workerID) {
70:     if (clients.length>0) {
71:       var nextClient = clients.shift();
72:       var msg = [workerID,'',nextclient.id,''].concat(nextClient.msg);
73:       sendToworker(msg);
74:       return true;
75:     } else
76:       return false;
77:   }
78:
79:   backend.on('message', function() {
80:     var args = Array.apply(null, arguments);
81:     if (args.length == 3) {
82:       if (!processPendingClient(args[0]))
83:         workers.push(args[0]);
84:     } else {
85:       var workerID = args[0];
86:       // Cancel the reply timeout.
87:       clearTimeout(busyWorkers[workerID].timeout);
88:       args = args.slice(2);
89:       frontend.send(args);
90:     if (!processPendingClient(workerID))
91:       workers.push(workerID);
92:     }
93:   });
```

3.2.3 Proves

Per a comprovar el correcte funcionament d'aquesta variant, haurem de comparar-ho amb el de la versió original, que pot ser complementada amb la seua extensió parametritzada. Per a fer això, iniciarem l'execució d'un procés *mybroker.js* i tres processos *myworker_vp.js*.

```
$ node mybroker &
[1] 12701
$ node myworker_vp 'tcp://*:8060' Worker1 id Answer1 &
[2] 12703
$ node myworker_vp 'tcp://*:8060' Worker2 id Answer2 &
[3] 12704
$ node myworker_vp 'tcp://*:8060' Worker3 id Answer3 &
[3] 12705
```

A continuació, eliminarem al primer d'aquests treballadors, mitjançant l'ordre kill, utilitzant com a argument el PID del procés (en el nostre exemple, apareix després del [2] i és el valor 12703):

```
$ kill 12703
```

Posteriorment, iniciarem tres clients d'aquesta manera:

```
$ node myclient & node myclient & node myclient &
```

Anota l'eixida obtinguda. Quantes respostes s'han obtingut? Quins treballadors les han enviades? Queda algun client esperant? (Pots utilitzar l'ordre **ps –o pid,args** per a esbrinar quins processos segueixen en marxa en la terminal).

Elimina tots els processos llançats en aquesta execució mitjançant l'ordre:

```
$ killall node
```

I ara repetirem la mateixa seqüència d'ordres, però utilitzant aquesta línia com a primera ordre de la seqüència:

```
$ node mybroker2 &
```

Després d'ella, seguirem llançant tres treballadors, per a eliminar posteriorment al primer d'ells. Després d'haver llançat tres clients... Quina eixida proporcionen ara? Quantes respostes han rebut? Quins treballadors les van generar? Per què? Queda algun client esperant?

Utilitzant *mybroker2.js*, si suposem que aquests treballadors són rèpliques d'un mateix servei i que tots generen les mateixes respostes, es proporcionaria transparència de replicació d'aquesta manera? Es proporcionaria transparència de fallades?

3.3 Equilibrat de càrrega

3.3.1 Punts clau del problema

Volem que el *broker* use un criteri de repartiment basat en la càrrega efectiva que tinguen els treballadors a cada moment.

- Recordem que es disposa d'una funció que retorna la càrrega de treball d'una màquina donada, anomenada ***getLoad***, el codi de la qual s'inclou en el mòdul *auxfunctions1718.js* (vegeu l'Annex 1).
- A més, convé llegir el punt 3 de la pràctica 1 (aplicació final: Proxy TCP/IP invers), entenent que els serveis remots ara són treballadors que hem de desenvolupar.

3.3.2 Punts clau de la solució

El codi del client no es veu afectat per aquesta modificació. El broker haurà de disposar d'una variable en la qual guarde les identitats de tots els treballadors, així com la seua disponibilitat.

En aquest cas la identitat del treballador ha de ser seleccionada explícitament pel broker d'entre la llista de treballadors disponibles i utilitzant un criteri d'equilibrat de càrrega. Aquesta identitat és necessària perquè el socket ROUTER puga col·locar el missatge en la cua d'enviament associada al treballador.

Sembla convenient que el broker estiga informat en tot moment del nivell d'activitat dels treballadors, però establir una periodicitat per a informar no sempre està justificat ni és útil. Al cap i a la fi, l'arribada de la petició d'un client és el desencadenant de l'elecció: tota la informació sobre la càrrega dels treballadors queda resumida en **la càrrega anunciada més recent entre els treballadors que esperen peticions**.

- Els treballadors no necessiten informar mentre processen peticions.
- Els treballadors que esperen, no modifiquen la seua càrrega[13] mentre ho fan. Per tant, la seua última notificació segueix sent vàlida.

[13] Es tracta d'una simplificació en la qual el treballador treballa en exclusiva per a aquest sistema

D'ací podem deduir que qualsevol treballador ha d'informar de la seua càrrega abans de passar a l'espera. Aquesta és una conclusió que ens ajudarà molt en la implementació.

En resum, la idea serà la següent:

- Els treballadors notificaran la seua càrrega efectiva en tot missatge que envien al *broker*. El *broker* prendrà nota d'aquesta càrrega i a qui correspon per a implantar el seu algorisme d'equilibrat de càrrega.

- En cada ocasió en la qual el broker haja d'assignar una petició a un treballador, seleccionarà entre els disponibles aquell que haja anunciat menor càrrega, li passarà la petició i el retirarà de la seua llista (el treballador deixa de ser elegible mentre atén la petició).

Podem trobar una dificultat instrumental per a dissenyar i implementar una llista eficient que represente als treballadors en funció de les seues càrregues. Les dues alternatives bàsiques són:

a) Mantenir un array no ordenat, amb inserció en posició arbitrària, i consumir temps per a trobar el treballador amb menor càrrega.
b) Mantenir un array ordenat, consumint temps per a inserir en ordre, i seleccionant sempre el primer treballador com el de menor càrrega.

Aplicant un criteri de senzillesa, en *auxfunctions1718.js* s'han col·locat les funcions *orderedList* (crear), *insert* (afegir), *lowest* (retirar element). Es basen en arrays i el mètode *sort* de JavaScript (https://www.w3schools.com/jsref/jsref_sort.asp).

- Pot reduir-se el nombre d'ocasions en què es necessite l'ordenació mitjançant un indicador (*ordered*) que es posa a fals cada vegada que s'afig, i a cert cada vegada que s'ordena. Les operacions d'addició sempre ho posen a fals, les d'extracció comproven que *ordered* siga cert (invocant *sort* en cas contrari) abans d'extraure l'element.

Desenvolupa les noves versions del treballador (`lbworker`) i de l'intermediari (`lbbroker`) que resolguen aquest problema. Comprova el seu funcionament: necessitaràs conèixer quines eleccions fa l'intermediari en cada ocasió, i si aquestes són les correctes.

Recorda el mode *verbose*!

4 ANNEX 1. FUNCIONS AUXILIARS

4.1.1 auxfunctions1718.js

```
01:   //auxfunctions1718
02:
03:     // *** getLoad function
04:
05:   function getLoad() {
06:     var fs     = require('fs')
07:       , data   = fs.readFileSync("/proc/loadavg") // synchronous
08:       , tokens = data.toString().split(' ')
09:       , min1   = parseFloat(tokens[0])+0.01
10:       , min5   = parseFloat(tokens[1])+0.01
11:       , min15  = parseFloat(tokens[2])+0.01
12:       , m      = min1*10 + min5*2 + min15;
13:     return m;
14:   }
15:
16:     // *** randNumber function
17:
18:   function randNumber(upper, extra) {
19:     var num = Math.abs(Math.round(Math.random() * upper));
20:     return num + (extra || 0);
21:   }
22:
23:     // *** randTime function
24:
25:   function randTime(n) {
26:     return Math.abs(Math.round(Math.random() * n)) + 1;
27:   }
28:
29:     // *** showMessage function
30:
31:   function showMessage(msg) {
32:     msg.forEach( (value,index) => {
33:       console.log( '     Segment %d: %s', index, value );
34:     })
35:   }
36:
37:     // *** ordered list functions for workers management
38:     // list has an "ordered" property (true|false) and a data property (array)
39:     // data elements consists of pairs {id, load}
40:
41:   function orderedList() {
42:     return {ordered: true, data: []};
43:   }
44:
45:   function nonempty() {
46:     return (this.data.length > 0);
47:   }
48:
49:   function lowest() {  // ordered reads
50:     if (!this.ordered) {
51:       this.data.sort(function(a, b) {
52:         return parseFloat(b.load) - parseFloat(a.load);
53:       }
54:     }
55:     this.ordered = true;
56:     return this.data.shift();
57:   }
58:
59:   function insert(k) { // unordered writes
60:     this.ordered = false;
61:     this.data.push(k);
```

```
62:     }
63:
64:     module.exports.getLoad = getLoad;
65:     module.exports.randNumber = randNumber;
66:     module.exports.randTime = randTime;
67:     module.exports.showMessage = showMessage;
68:     module.exports.orderedList = orderedList;
69:     module.exports.nonempty = nonempty;
70:     module.exports.lowest = lowest;
71:     module.exports.insert = insert;
```

5 ANNEX 2. EL MODE *VERBOSE*

Mostrem aquestes execucions amb la traça d'algunes combinacions significatives dels agents. S'exclou el client perquè a penes aporta informació.

En aquests dos casos cap client ni cap treballador espera que l'anterior finalitze:

- El primer ordre d'execució mostrat (***bc1c2c4c3w2w1***) és broker, client1, client2, client4, client3, worker2, worker1.
- El segon ordre (***bw1w2c1c2c3c4***) és broker, worker1, worker2, client1, client2, client3, client4.

5.1.1 Execució de mybroker_vp.js en mode *verbose*

```
node mybroker_vp.js 8059 8060 verbose
```

Ordre *bc1c2c4c3w2w1*, pantalla del broker

```
broker: frontend-router listening on tcp://*:8059
broker: backend-router listening on tcp://*:8060
Received request: "Hello1" from client (Client1).
     Segment 0: Client1
     Segment 1:
     Segment 2: Hello1
Pushing client (Client1) to clients' queue (size:
1).
Received request: "Hello2" from client (Client2).
     Segment 0: Client2
     Segment 1:
     Segment 2: Hello2
Pushing client (Client2) to clients' queue (size:
2).
Received request: "Hello4" from client (Client4).
     Segment 0: Client4
     Segment 1:
     Segment 2: Hello4
Pushing client (Client4) to clients' queue (size:
3).
Received request: "Hello3" from client (Client3).
     Segment 0: Client3
     Segment 1:
     Segment 2: Hello3
Pushing client (Client3) to clients' queue (size:
4).
Received backend request: "id" from worker
(Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: id
Sending client (Client1) request to worker
(Worker2) through backend.
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client1
     Segment 3:
     Segment 4: Hello1
Received backend request: "id" from worker
(Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: id
Sending client (Client2) request to worker
(Worker1) through backend.
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client2
     Segment 3:
     Segment 4: Hello2
Received reply: "World2" from worker (Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client1
     Segment 3:
     Segment 4: World2
Sending worker (Worker2) reply to client (Client1)
through frontend.
     Segment 0: Client1
     Segment 1:
     Segment 2: World2
Sending client (Client4) request to worker
(Worker2) through backend.
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client4
     Segment 3:
     Segment 4: Hello4
Received reply: "World1" from worker (Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client2
     Segment 3:
     Segment 4: World1
Sending worker (Worker1) reply to client (Client2)
through frontend.
     Segment 0: Client2
     Segment 1:
     Segment 2: World1
Sending client (Client3) request to worker
(Worker1) through backend.
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client3
     Segment 3:
     Segment 4: Hello3
Received reply: "World2" from worker (Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client4
     Segment 3:
     Segment 4: World2
Sending worker (Worker2) reply to client (Client4)
through frontend.
     Segment 0: Client4
     Segment 1:
     Segment 2: World2
Pushing worker (Worker2) to workers' queue (size:
1).
Received reply: "World1" from worker (Worker1)
     Segment 0: Worker1
     Segment 1:
```

```
     Segment 2: Client3
     Segment 3:
     Segment 4: World1
Sending worker (Worker1) reply to client (Client3)
through frontend.
     Segment 0: Client3
     Segment 1:
     Segment 2: World1
Pushing worker (Worker1) to workers' queue (size:
2).
```

Ordre *bw1w2c1c2c3c4*, pantalla del broker

```
broker: frontend-router listening on tcp://*:8059
broker: backend-router listening on tcp://*:8060
Received backend request: "id" from worker
(Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: id
Pushing worker (Worker1) to workers' queue (size:
1).
Received backend request: "id" from worker
(Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: id
Pushing worker (Worker2) to workers' queue (size:
2).
Received request: "Hello1" from client (Client1).
     Segment 0: Client1
     Segment 1:
     Segment 2: Hello1
Sending client (Client1) request to worker
(Worker1) through backend.
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client1
     Segment 3:
     Segment 4: Hello1
Received reply: "World1" from worker (Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client1
     Segment 3:
     Segment 4: World1
Sending worker (Worker1) reply to client (Client1)
through frontend.
     Segment 0: Client1
     Segment 1:
     Segment 2: World1
Pushing worker (Worker1) to workers' queue (size:
2).
Received request: "Hello2" from client (Client2).
     Segment 0: Client2
     Segment 1:
     Segment 2: Hello2
Sending client (Client2) request to worker
(Worker2) through backend.
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client2
     Segment 3:
     Segment 4: Hello2
Received reply: "World2" from worker (Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client2
     Segment 3:
     Segment 4: World2
Sending worker (Worker2) reply to client (Client2)
through frontend.
     Segment 0: Client2
     Segment 1:
     Segment 2: World2
Pushing worker (Worker2) to workers' queue (size:
2).
Received request: "Hello3" from client (Client3).
     Segment 0: Client3
     Segment 1:
     Segment 2: Hello3
Sending client (Client3) request to worker
(Worker1) through backend.
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client3
     Segment 3:
     Segment 4: Hello3
Received reply: "World1" from worker (Worker1)
     Segment 0: Worker1
     Segment 1:
     Segment 2: Client3
     Segment 3:
     Segment 4: World1
Sending worker (Worker1) reply to client (Client3)
through frontend.
     Segment 0: Client3
     Segment 1:
     Segment 2: World1
Pushing worker (Worker1) to workers' queue (size:
2).
Received request: "Hello4" from client (Client4).
     Segment 0: Client4
     Segment 1:
     Segment 2: Hello4
Sending client (Client4) request to worker
(Worker2) through backend.
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client4
     Segment 3:
     Segment 4: Hello4
Received reply: "World2" from worker (Worker2)
     Segment 0: Worker2
     Segment 1:
     Segment 2: Client4
     Segment 3:
     Segment 4: World2
Sending worker (Worker2) reply to client (Client4)
through frontend.
     Segment 0: Client4
     Segment 1:
     Segment 2: World2
Pushing worker (Worker2) to workers' queue (size:
2).
```

5.1.2 Execució de myworker_vp.js en mode *verbose*

```
node myworker_vp.js tcp://localhost:8060 Worker1 id World1 verbose
```

Ordre *bc1c2c4c3w2w1*, pantalla de worker1

```
Worker (worker1) connected to tcp://localhost:8060
Worker (worker1) has sent its first connection message: "id"
Worker (worker1) has received request "Hello2" from client (Client2)
Worker (worker1) has sent its reply "World1"
Worker (worker1) has received request "Hello3" from client (Client3)
Worker (worker1) has sent its reply "World1"
```

Ordre *bw1w2c1c2c3c4*, pantalla de worker1

```
Worker (worker1) connected to tcp://localhost:8060
Worker (worker1) has sent its first connection message: "id"
Worker (worker1) has received request "Hello1" from client (Client1)
Worker (worker1) has sent its reply "World1"
Worker (worker1) has received request "Hello3" from client (Client3)
Worker (worker1) has sent its reply "World1"
```

6 ANNEX: CODIS FONT

6.1 Codi COMENTAT dels components

6.1.1 myclient.js

```
01:  // myclient in NodeJS
02:  // By default:
03:  // - It connects REQ socket to tcp://localhost:8059
04:  // - It sends "Hello" to server and expects "World" back
05:  // Using the command line arguments, that default behaviour
06:  // may be changed. To this end, those arguments are interpreted
07:  // as follows:
08:  // - 1st: URL of the broker frontend socket.
09:  // - 2nd: Client ID, to tag the connection to the frontend router
10:  //        socket of the broker.
11:  // - 3rd: String to be sent to the servers (i.e., worker agents).
12:
13:  var zmq       = require('zmq')
14:    , requester = zmq.socket('req');
15:
16:  // Command-line arguments.
17:  var args = process.argv.slice(2);
18:  // Get those arguments.
19:  var brokerURL = args[0] || 'tcp://localhost:8059';
20:  var myID = args[1] || 'NONE';
21:  var myMsg = args[2] || 'Hello';
22:
23:  // Set the connection ID. This must be done before the
24:  // connect() method is called.
25:  if (myID != 'NONE')
26:        requester.identity = myID;
27:  // Connect to the frontend ROUTER socket of the broker.
28:  requester.connect(brokerURL);
29:  // Print trace information.
30:  console.log('Client (%s) connected to %s', myID, brokerURL)
31:
32:  // A single reply is expected...
33:  requester.on('message', function(msg) {
34:    console.log('Client (%s) has received reply "%s"', myID, msg.toString());
35:    process.exit(0);
36:  });
37:
38:  // Send a single message.
39:  requester.send(myMsg);
40:  // Print trace information.
41:  console.log('Client (%s) has sent its message: "%s"', myID, myMsg);
```

6.1.2 myworker.js

```
01:  // myworker server in NodeJS
02:  // By default:
03:  // - It connects its REQ socket to tcp://*:8060
04:  // - It expects a request message from client, and it replies with "World"
05:  // Multiple non-mandatory command-line arguments can be provided in
06:  // the following order:
07:  // - 1st: URL of the broker backend socket.
08:  // - 2nd: Worker identifier (a string).
09:  // - 3rd: Text to be used in its initial connection message.
10:  // - 4th: Text to be used in the replies to be returned to clients.
11:
12:  var zmq = require('zmq')
```

```
13:     , responder = zmq.socket('req');
14:
15:   // Get the command-line arguments, if any.
16:   var args = process.argv.slice(2);
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:   // Set the worker identity to the connection.
23:   // Note that a random ID is assigned by default.
24:   // Thus, when no command-line arguments are given,
25:   // no identity must be set.
26:   if (myID != 'NONE')
27:         responder.identity = myID;
28:   // Connect to the broker backend socket.
29:   responder.connect(backendURL);
30:
31:   // Process each incoming request.
32:   responder.on('message', function(client, delimiter, msg) {
33:         setTimeout(function() {
34:               responder.send([client,'',replyText]);
35:         }, 1000);
36:   });
37:
38:   // This is the first message sent by a worker.
39:   // Since this "responder" socket has been connected
40:   // to a ROUTER socket, ZeroMQ will prepend the identity
41:   // of the worker to every sent message. That identity
42:   // is used by the broker to "register" this
43:   // worker in its appropriate data structures.
44:   responder.send(connText);
```

6.1.3 mybroker.js

```
01:   // ROUTER-ROUTER request-reply broker in Node.js
02:   // It uses a FIFO policy to administer its pending
03:   // clients and available workers.
04:   // Each kind of agent is placed in its corresponding
05:   // array, that is managed as a queue by default.
06:
07:   // - 1st: Port number for its frontend socket (8059)
08:   // - 2nd: Port number for its backend socket (8060)
09:
10:   var zmq      = require('zmq')
11:     , frontend = zmq.socket('router')
12:     , backend  = zmq.socket('router');
13:
14:   // Get the command-line arguments.
15:   var args = process.argv.slice(2);
16:   // Port number for the frontend socket.
17:   var fePortNbr = args[0] || 8059;
18:   // Port number for the backend socket.
19:   var bePortNbr = args[1] || 8060;
20:
21:   // Array of available workers.
22:   var workers = [];
23:   // Array of pending clients.
24:   var clients = [];
25:
26:   frontend.bindSync('tcp://*:'+fePortNbr);
27:   backend.bindSync('tcp://*:'+bePortNbr);
28:
29:   frontend.on('message', function() {
30:     // Note that separate message parts come as function arguments.
```

```
31:     var args = Array.apply(null, arguments);
32:     // Check whether there is any available worker.
33:     if (workers.length > 0) {
34:       // Remove the oldest worker from the array.
35:       var myWorker = workers.shift();
36:       // Build a multi-segment message.
37:       // & Send it.
38:       var m = [myWorker,''].concat(args);
39:       backend.send(m);
40:     } else
41:       // When no available worker exists, save
42:       // the client ID and message into the clients array.
43:       clients.push( {id: args[0],msg: args.slice(2)});
44:   });
45:
46:   function processPendingClient(workerID) {
47:     // Check whether there is any pending client.
48:     if (clients.length>0) {
49:       // Get the data from the first client and remove it
50:       // from the queue of pending clients.
51:       var nextClient = clients.shift();
52:       // Build the message and send it.
53:       // Note that a message is an array of segments. Therefore,
54:       // we should prepend: (a) the worker ID + a delimiter,
55:       // to choose the worker connection to be used, (b) the client ID +
56:       // a delimiter, to build the needed context for adequately
57:       // forwarding its subsequent reply at the worker domain.
58:       // & Send the message.
59:       var m = . [workerID,'',nextclient.id,''].concat(nextClient.msg);
60:       backend.send(m);
61:       // Return true if any client has been found
62:       return true;
63:     } else
64:       // Return false if no client is there.
65:       return false;
66:   }
67:
68:   backend.on('message', function() {
69:     var args = Array.apply(null, arguments);
70:     // If this is an initial ID message from a new worker,
71:     // save its identity in the "workers" array...
72:     if (args.length == 3) {
73:       // It may happen that some clients are already waiting
74:       // for available workers. If so, serve one of them!
75:       if (!processPendingClient(args[0]))
76:         // Otherwise, save this worker as an available one.
77:         workers.push(args[0]);
78:     // ...otherwise, return the message to the appropriate
79:     // client.
80:     } else {
81:     // Save the worker identity.
82:     var workerID = args[0];
83:     // Remove the first two slots (id+delimiter)
84:     // from the array.
85:     args = args.slice(2);
86:     // Send the resulting message to the appropriate
87:     // client. Its first slot holds the client connection
88:     // ID.
89:     frontend.send(args);
90:     // Check whether any client is waiting for attention.
91:     // If so, the request from the first one is sent and
92:     // it is removed from the pending queue.
93:     // Otherwise, a false value is returned.
94:     if (!processPendingClient(workerID))
95:       // In that case, the worker ID is saved in the
96:       // queue of available workers.
```

```
97:       workers.push(workerID);
98:     }
99:   });
```