TSR - PRÁCTICA 2
CURSO 2014/15

ØMQ

Esta práctica, que consta de tres sesiones, tiene como propósito:

> "Entrenar al alumno en el uso del sistema de mensajería de colas(ØMQ). Al finalizar esta parte de laboratorio, el alumno debe conocer cómo desarrollar aplicaciones con 0MQ sobre node.js"

Para cumplir con este propósito, establecemos objetivos según el siguiente guión:

Los alumnos profundizarán en los patrones básicos de comunicación en ØMQ, que ya han visto en las clases de seminario, realizando diversas modificaciones en programas dados que implementen dichos patrones.

Los alumnos implementarán un repartidor de mensajes de petición de servicio:

- En este sistema, habrá un servidor (llamado ***proxy*** o ***broker***) que expone un ***socket* ØMQ** del tipo **ROUTER**, aceptando conexiones remotas de clientes.
- Adicionalmente, el servidor tendrá conexiones con un ***pool*** de trabajadores dispuestos a gestionar las peticiones de los clientes. Se ha de usar la mejor forma de establecer estas conexiones de entre las posibles facilitadas por ØMQ.
- Cuando llega un mensaje por el socket de entrada, el broker selecciona uno de los trabajadores, y le pasa el mensaje.
- Cuando el trabajador termina de gestionar el mensaje, le envía un mensaje de respuesta al servidor, quien lo reenvía de vuelta a quien le envió la petición original.

Finalmente, los alumnos implementarán algunas modificaciones en este sistema repartidor de mensajes de peticiones de servicio:

- Introducir el uso de **promesas** en la gestión de las respuestas de los trabajadores.
- Modificar, en el broker, el **criterio de reparto** equitativo de la carga entre los trabajadores por otro criterio basado en la carga efectiva que tengan éstos.
- Añadir al proxy lo necesario para que pueda **detectar** si los trabajadores han dejado de funcionar. Debería poder verificar periódicamente si sus **trabajadores** están **"vivos"**.
- Añadir al broker lo necesario para que se pueda **configurar dinámicamente**. Para ello, presentará un socket ØMQ de respuesta, que será usado desde un configurador para determinar los criterios de routing.
- Para definir la interfaz de llamada de configuración se usarán **especificaciones en JSON** (estructura de los mensajes de petición y de contestación).

En este boletín, el **apartado 1** describe las actividades relativas al primer objetivo (estudio de los patrones básicos de comunicación en ØMQ), el **apartado 2** aborda el segundo objetivo (implementación del repartidor de mensajes de petición de servicio), y el **apartado 3** presenta las modificaciones a realizar sobre el sistema implementado en el apartado anterior. A continuación, se encuentran los detalles relativos a la entrega de la práctica, así como los anexos con información de utilidad.

En cuanto a ordenación temporal, está previsto que el apartado 1 se cubra en la primera sesión, que el apartado 2 se desarrolle en la segunda sesión, y que el apartado 3 se complete en la tercera sesión.

El material producido durante esta práctica se mantendrá creando un proyecto, **lab2**, mantenido utilizando depósitos git, siguiendo el flujo de trabajo establecido en la práctica 1.

CONTENIDO

1 APARTADO 0. INTRODUCCIÓN A ØMQ (REPASO)

En las actividades de laboratorio se va a utilizar el *middleware* de comunicaciones basado en colas, llamado **ØMQ** o **ZeroMQ**, versión 3.x.x, que ya está instalado en el laboratorio. Sin embargo, para poder usarlo en conjunción con nuestro lenguaje de programación asíncrona, **Node.js**, hay que instalar (en el $HOME del usuario) los *bindings* correspondientes a dicho lenguaje. Para ello, desde una terminal, se tiene que ejecutar la siguiente orden:

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

La ejecución correcta de este comando genera un directorio `$HOME/node_modules/zmq` que contiene todo lo necesario para usar ØMQ en programas de Node.js.

A partir de ese momento, los programas que incluyan / que requieran ØMQ…

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

…no tendrán problemas para acceder a la biblioteca de ØMQ. Si, al ejecutar un programa que incluya la anterior línea de código, se genera un mensaje de error similar al siguiente…

```
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)
    …
```

…es señal de que no se ha ejecutado correctamente la orden `npm install zmq`.

Se dispone de información completa sobre ØMQ en su *website* oficial: http://zeromq.org. En particular, en cuanto a estudio y documentación, son esenciales:

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

En esta documentación, C es el lenguaje de programación usado como referencia en los ejemplos. Además, también se incluyen algunos ejemplos programados en otros lenguajes (C++, Java, Haskell, Node.js, Python, etc.).

En "ØMQ – The Guide" se informa de la existencia de un depósito Git público que contiene todos los ejemplos de la guía. Cualquiera puede clonar dicho depósito mediante la ejecución de `git clone` especificando la dirección del depósito (el comando exacto está disponible en la guía). Sin embargo, no clonaremos ese depósito, dado que incluye muchos ficheros que no usaremos (ejemplos en otros lenguajes).

En la asignatura TSR, hemos preparado un depósito Git público (accesible para todos los usuarios de la asignatura) que contiene los ejemplos de "ØMQ – The Guide" escritos en

Node.js. Podemos clonar dicho depósito (y esto es nuestra primera actividad en esta práctica) mediante:

```
bash-4.1$ git clone https://gitlab.dsic.upv.es/tsr/zmq_nodejs_examples.git
```

En el siguiente apartado de esta práctica, estudiaremos y modificaremos varios de los ejemplos incluidos en el depósito ***zmq_nodejs_examples***. Dichos ejemplos son implementaciones correctas de varios patrones básicos de comunicación con ØMQ. Sin embargo, la ejecución de los mismos, sin modificar nada, genera errores en las máquinas de los laboratorios dado que se usan puertos que no tenemos disponibles. Conviene recordar, una vez más, que se dispone, exclusivamente, de los **puertos 8000 a 8009**, ambos inclusive.

2 APARTADO 1. PROGRAMACIÓN BÁSICA CON ØMQ

Una vez dispongamos de nuestro clon de ***zmq_nodejs_examples***, se llevarán a cabo varias modificaciones en algunos de los programas ejemplo. Para ello, se ha de crear un nuevo directorio bajo **lab2**, llamado ***zmqbasico***, al que se añadirán los programas ejemplo que sean objeto de modificación.

2.1 Estudio del patrón REQ - REP

Para empezar, vamos a estudiar dos programas, cuyos códigos se reproducen a continuación, que muestran la interacción más sencilla posible con los *sockets* ØMQ:

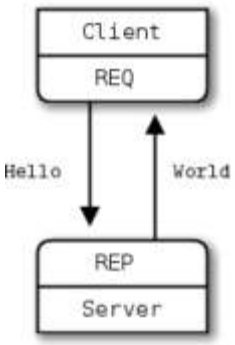
2.1.1 hwclient.js

```
01:  // Hello World client
02:  // Connects REQ socket to tcp://localhost:5555
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:5555");
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:  });
```

2.1.2 hwserver.js

```
01:  // Hello World server
02:  // Binds REP socket to tcp://*:5555
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://*:5555', function(err) {
22:    if (err) {
23:      console.log(err);
24:    } else {
25:      console.log("Listening on 5555...");
26:    }
27:  });
28:
29:  process.on('SIGINT', function() {
30:    responder.close();
31:  });
```

En estos dos programas se implementa el patrón **REQ** - **REP** de la comunicación con sockets ØMQ. Tras leer y comprender su código, se realizarán las siguientes modificaciones (en el proyecto ***zmqbasico***):



1) El programa cliente tendrá los siguientes argumentos:
    a. URL del *endpoint* en el servidor[1].
    b. Número de peticiones a enviar.
    c. Texto de la petición a enviar (que podría incluir el carácter espacio en blanco).

2) El programa servidor tendrá los siguientes argumentos:
    a. Puerto del servidor.
    b. Número de segundos a esperar en cada respuesta.
    c. Texto para la respuesta a enviar (que podría incluir el carácter espacio en blanco).

[1] La URL de un determinado *endpoint* consta del nombre o dirección IP del mismo, y el número de *port*.

3) La respuesta que el servidor enviará al cliente será la concatenación de la petición del cliente y el texto de respuesta propia del servidor (el texto del último argumento en su invocación).

4) Si cualquiera de los programas no recibiera el número correcto de argumentos, deberá mostrar un mensaje que informe sobre su invocación correcta, y concluir su ejecución.

Cliente y servidor deberían poder ejecutarse en una misma máquina (*localhost*) o en dos máquinas diferentes. El servidor también debe poder atender peticiones de varios clientes: observar el comportamiento del servidor en cuanto al orden de atención de peticiones concurrentes de clientes diferentes[2].

Las nuevas versiones de ***hwclient.js*** y ***hwserver.js*** deben ser objeto de actualización en los depósitos local y remoto (`git commit`, `git push`) asociados al proyecto ***zmqbasico***, al menos una vez, y en operaciones hechas por distintos miembros del equipo.

2.2 Estudio del patrón PUB - SUB

El siguiente paso es estudiar otros dos programas, disponibles en ***zmq_nodejs_examples***, cuyos códigos se reproducen a continuación, que muestran otro uso relevante de los sockets ØMQ:

2.2.1 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:8688")
09:  subscriber.subscribe("")
10:
11:  process.on('SIGINT', function() {
12:    subscriber.close()
13:    console.log('\nClosed')
14:  })
```

2.2.2 publisher.js

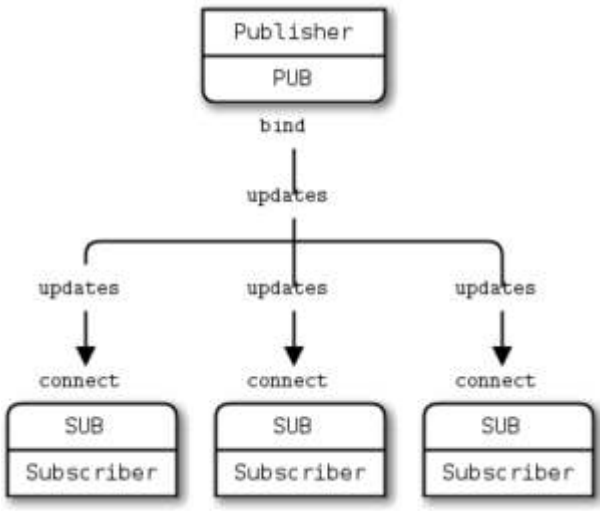
```
01:  var zmq = require('zmq')
02:  var publisher = zmq.socket('pub')
03:
04:  publisher.bind('tcp://*:8688', function(err) {
05:    if(err)
06:      console.log(err)
07:    else
08:      console.log("Listening on 8688...")
09:  })
```

[2] Recordar el patrón básico REQ-REP en el caso de múltiples peticionarios, visto en el Seminario 3.

```
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:   })
```

En estos dos programas se implementa el patrón **PUB** - **SUB** de la comunicación con sockets ØMQ. Tras leer y comprender su código, se realizarán las siguientes modificaciones (en el proyecto ***zmqbasico***):

1) El programa subscriptor tendrá los siguientes argumentos:
    a. URL del *endpoint* en el publicador.
    b. Descriptor del tipo de mensajes al que se subscribe.



2) El programa publicador tendrá los siguientes argumentos:
    a. Puerto del publicador.
    b. Número de mensajes a publicar.
    c. Descriptores de dos tipos de mensajes a publicar.

3) El publicador publicará el mismo número de mensajes de los dos tipos especificados. Cada mensaje publicado consistirá en la concatenación del descriptor de tipo de mensaje y un número aleatorio[3].

4) Los subscriptores se conectarán al publicador, especificando su URL, así como el descriptor del tipo de mensaje que les interese recibir (filtro de mensajes).

Hay que comprobar el funcionamiento del patrón PUB-SUB, ejecutando los programas en una misma máquina (*localhost*) o en dos máquinas diferentes. Al menos, se debe verificar la conexión de dos subscriptores con distintos filtros de mensajes a un mismo publicador.

[3] Para obtener números aleatorios, en un rango determinado, puede usarse la función ***randNumber***, importando el módulo ***auxfunctions.js*** (cuyo código se facilita en el *Anexo 1. Funciones auxiliares*).

Las nuevas versiones de ***subscriber.js*** y ***publisher.js*** deben ser objeto de actualización en los depósitos local y remoto (`git commit`, `git push`) asociados al proyecto ***zmqbasico***, al menos una vez, y en operaciones hechas por distintos miembros del equipo.

2.3 Estudio del patrón ROUTER - DEALER

Para terminar ***zmqbasico***, se van a estudiar los siguientes programas, disponibles en ***zmq_nodejs_examples***:

2.3.1 rrclient.js

```
01:  // Hello World client in Node.js
02:  // Connects REQ socket to tcp://localhost:5559
03:  // Sends "Hello" to server, expects "World" back
04:
05:  var zmq       = require('zmq')
06:    , requester = zmq.socket('req');
07:
08:  requester.connect('tcp://localhost:5559');
09:  var replyNbr = 0;
10:  requester.on('message', function(msg) {
11:    console.log('got reply', replyNbr, msg.toString());
12:    replyNbr += 1;
13:  });
14:
15:  for (var i = 0; i < 10; ++i) {
16:    requester.send("Hello");
17:  }
```

2.3.2 rrworker.js

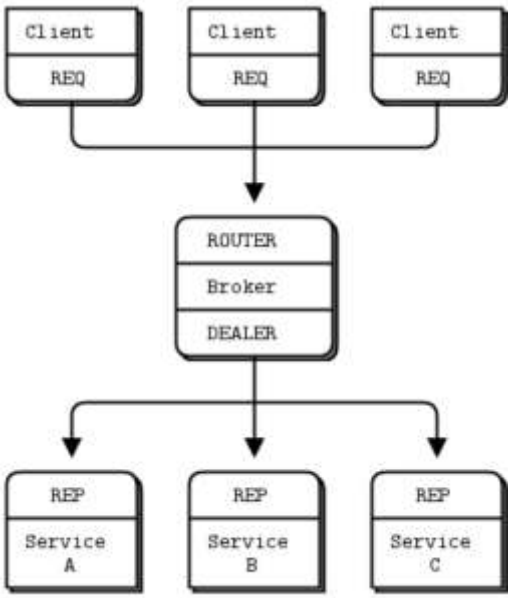
```
01:  // Hello World server in Node.js
02:  // Connects REP socket to tcp://*:5560
03:  // Expects "Hello" from client, replies with "World"
04:
05:  var zmq = require('zmq')
06:    , responder = zmq.socket('rep');
07:
08:  responder.connect('tcp://localhost:5560');
09:  responder.on('message', function(msg) {
10:    console.log('received request:', msg.toString());
11:    setTimeout(function() {
12:      responder.send("World");
13:    }, 1000);
14:  });
```

2.3.3 rrbroker.js

```
01:  // Simple request-reply broker in Node.js
02:
03:  var zmq      = require('zmq')
04:    , frontend = zmq.socket('router')
05:    , backend  = zmq.socket('dealer');
06:
07:  frontend.bindSync('tcp://*:5559');
08:  backend.bindSync('tcp://*:5560');
09:
10:  frontend.on('message', function() {
11:    // Note that separate message parts come as function arguments.
```

```
12:     var args = Array.apply(null, arguments);
13:     // Pass array of strings/buffers to send multipart messages.
14:     backend.send(args);
15:   });
16:
17:   backend.on('message', function() {
18:     var args = Array.apply(null, arguments);
19:     frontend.send(args);
20:   });
```

En estos tres programas se implementa el patrón **ROUTER** - **DEALER** de la comunicación con sockets ØMQ, en combinación con **REQ**, para las peticiones de los clientes, y **REP**, para las respuestas de los trabajadores o servidores. Tras leer y comprender su código, se realizarán las siguientes modificaciones (en el proyecto ***zmqbasico***):



1) El programa cliente (*rrclient.js*) tendrá los siguientes argumentos:
    a. URL del *endpoint* del ROUTER en el broker.
    b. Número de peticiones a enviar.
    c. Texto de la petición a enviar (que podría incluir el carácter espacio en blanco).

2) El programa servidor (*rrworker.js*) tendrá los siguientes argumentos:
    a. URL del *endpoint* del DEALER en el broker.
    b. Número de segundos a esperar en cada respuesta.
    c. Texto para la respuesta a enviar (que podría incluir el carácter espacio en blanco).

3) El programa intermediario (*rrbroker.js*) tendrá los siguientes argumentos:
    a. Puerto de comunicación con los clientes (ROUTER).
    b. Puerto de comunicación con los servidores (DEALER).

4) Un cliente terminará su ejecución cuando reciba respuesta a todas sus peticiones.

5) Los servidores y el intermediario (broker) no terminarán su ejecución, seguirán a la espera de peticiones de nuevos clientes.

Hay que comprobar el funcionamiento del patrón ROUTER-DEALER, ejecutando los programas en una misma máquina (*localhost*) o en dos máquinas diferentes. Al menos, se debe verificar:

1) El funcionamiento con un cliente y dos servidores. Para ello, indicar al ejecutar los servidores, en su último argumento (respuesta a enviar): "*I am server one*", "*I am server two*", respectivamente[4]. Comprobad el reparto de trabajo en la atención de las peticiones del cliente.

2) El funcionamiento con dos clientes y un servidor. Para ello, indicar al ejecutar los clientes, en su último argumento (petición a enviar): "*I am client one*", "*I am client two*", respectivamente. Comprobad si el servidor atiende equitativamente las peticiones de los clientes.

3) El funcionamiento con dos clientes y tres servidores, indicando también mensajes identificativos tanto de clientes como de servidores. Comprobad si quedan peticiones no atendidas, así como el reparto de trabajo entre los servidores.

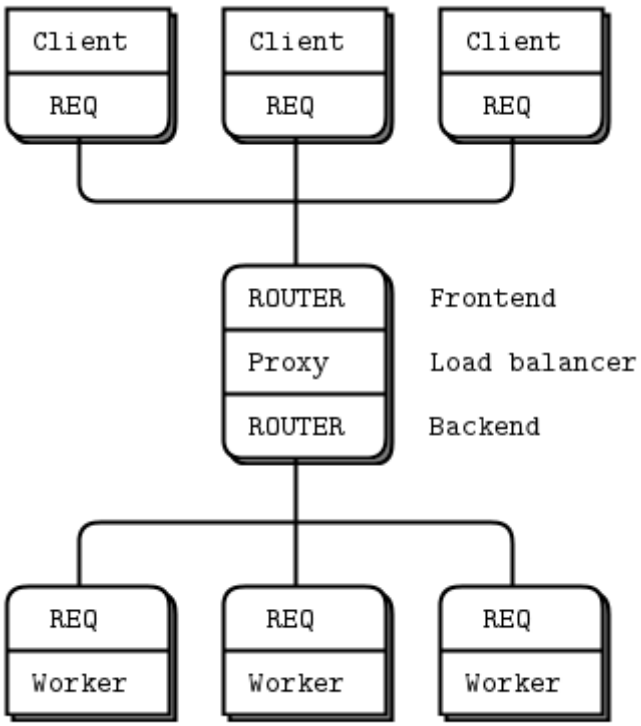
Las nuevas versiones de ***rrclient.js***, ***rrworker.js*** y ***rrbroker.js*** deben ser objeto de actualización en los depósitos local y remoto (`git commit`, `git push`) asociados a ***zmqbasico***, al menos una vez, y en operaciones hechas por distintos miembros del equipo.

[4] En el siguiente apartado (proyecto) de esta práctica se planteará el modo adecuado para que clientes y trabajadores establezcan su identidad al conectarse a sockets ROUTER de un broker. Consideraremos aceptable, en este punto del aprendizaje, la identificación simplista mediante el propio texto del mensaje (petición o respuesta): "*I am client …*", "*I am server …*".

3 APARTADO 2. PROGRAMACIÓN AVANZADA CON ØMQ

3.1 Introducción al Proxy con ØMQ

El objetivo de este apartado es la implementación de un proxy que gestione mensajes de petición de servicio. Este proxy, intermediario o ***broker***, tendrá la siguiente funcionalidad:



- Presentará un **socket ØMQ** de tipo **ROUTER**, donde aceptará conexiones remotas de los clientes (quienes le enviarán mensajes de petición de servicio). Llamemos ***frontend*** a este socket.
- Presentará otro **socket ØMQ** de tipo **ROUTER**, donde aceptará conexiones remotas de los servidores o trabajadores (quienes atenderán las peticiones de servicio que les lleguen). Llamemos ***backend*** a este socket.
- Cuando un cliente, desde su **socket ØMQ** de tipo **REQ**, envíe una petición de servicio (mensaje) hacia el socket *frontend*, el proxy seleccionará un trabajador, y le pasará el mensaje por el socket *backend*.
- La selección del trabajador al que se le envíe la petición de servicio se realizará aplicando algún criterio de equilibrado de carga (*load balancing pattern*).
- Una vez se atienda la petición de servicio (el mensaje), el trabajador que la haya atendido lo comunica, desde su **socket ØMQ** de tipo **REQ**, mediante una petición enviada al proxy, por el socket *backend*.

- Cuando llegue esta petición[5] de un trabajador por el socket *backend*, el proxy la ha de reenviar, por el socket *frontend*, al cliente que corresponda (al que envió la petición original).
- La gestión de los mensajes descrita exige que el proxy conozca las identidades de clientes y trabajadores.

Antes de implementar nada, se ha de crear un nuevo directorio, llamado ***zmqavanzado***, al que se añadirán los programas necesarios para poner en funcionamiento el esquema descrito. Los programas a implementar se llamarán: ***lbbroker.js***, ***lbworker.js*** y ***lbclient.js***.

Antes de implementar nada, hay que leer detenidamente las explicaciones que siguen acerca de cada uno de estos programas.

Por lo demás, cuando llegue el momento de implementar los programas, conforme se tengan nuevas versiones de los mismos (sean incompletas, sean completas aunque con problemas pendientes de resolución) se ha de seguir la forma de trabajo habitual con Git, actualizando los correspondientes depósitos local y remoto con las sucesivas versiones de los programas. No se insistirá más en este punto relativo a la metodología de las prácticas de TSR.

3.2 Descripción del cliente

El programa ***lbclient.js*** contendrá el código necesario para ejecutar un cliente (conectado al *frontend* del proxy). Se puede usar, como punto de partida, el programa *rrclient.js* desarrollado en el proyecto anterior. La funcionalidad del cliente será la siguiente:

1) Tendrá los siguientes argumentos:
    a. URL del *endpoint* del ROUTER *frontend* en el broker (proxy).
    b. Cadena de caracteres (*string*) que represente la identidad del cliente.
    c. Texto de la petición de servicio, por ejemplo: '*work_harder*'.

2) El cliente usará un socket **REQ** para enviar su mensaje de petición de servicio. Sólo enviará una petición.

3) El cliente quedará a la espera de recibir la respuesta a su petición, cuando ésta llegue el cliente terminará su ejecución.

3.3 Descripción del trabajador (servidor)

[5] Esta petición del trabajador la podemos interpretar, de un modo informal, como la respuesta que se da a la petición de servicio que envió el cliente.

El programa ***lbworker.js*** contendrá el código necesario para ejecutar un trabajador (conectado al *backend* del proxy). La funcionalidad del trabajador será la siguiente:

1) Tendrá los siguientes argumentos:

    a. URL del *endpoint* del ROUTER *backend* en el broker (proxy).
    b. Cadena de caracteres (*string*) que represente la identidad del trabajador.
    c. Texto de un mensaje de disponibilidad[6], por ejemplo: '*I_am_ready*'.
    d. Texto de un mensaje de atención de servicio[7], por ejemplo: '*work_done*'.
    e. Un valor (*true* / *false*) que indicará si se activa el modo *verbose* (que proporcionará información detallada en la salida de consola).

2) El trabajador usará un socket **REQ** para comunicarse con el broker:

    a. Sólo enviará una petición para darse de alta en el broker (al iniciarse, informando de su disponibilidad).
    b. Enviará, por cada petición de servicio que reciba, y una vez ésta sea atendida, una petición al broker para que notifique al cliente correspondiente que el servicio ha sido atendido.

3) El trabajador quedará en ejecución indefinida: siempre a la espera de recibir peticiones de servicio.

3.4 Descripción del broker (proxy) y de los mensajes

El programa ***lbbroker.js*** contendrá el código necesario para ejecutar el proxy, intermediario o ***broker***. Su funcionalidad ya se ha descrito al inicio de este apartado, pero es momento de presentar algunas características más:

1) Tendrá los siguientes argumentos:

    a. Sus dos puertos (el de *frontend*, y el de *backend*).
    b. Un valor (*true* / *false*) que indicará si se activa el modo *verbose* (que proporcionará información detallada en la salida de consola).

2) Como es obvio, usará 2 sockets **ROUTER** para gestionar las comunicaciones.

3) Almacenará en alguna estructura de datos interna (cola, lista, tabla…) los identificadores de los trabajadores que le hayan enviado mensajes de disponibilidad.

4) Cuando reciba peticiones de servicio (en su *frontend*), verificará si dispone de trabajadores que puedan atenderlo y, en tal caso, seleccionará al más adecuado (en

[6] Mensaje de petición de darse de alta en el broker.

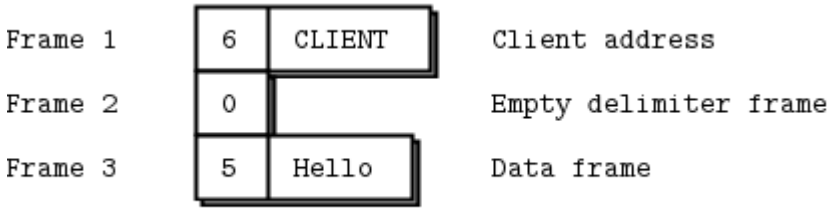
[7] Mensaje de petición al broker para notificar a un cliente que se le ha prestado el servicio.

términos de equilibrio de carga), reenviándole la petición (a través de su *backend*). En caso contrario, guardará la petición en una cola.

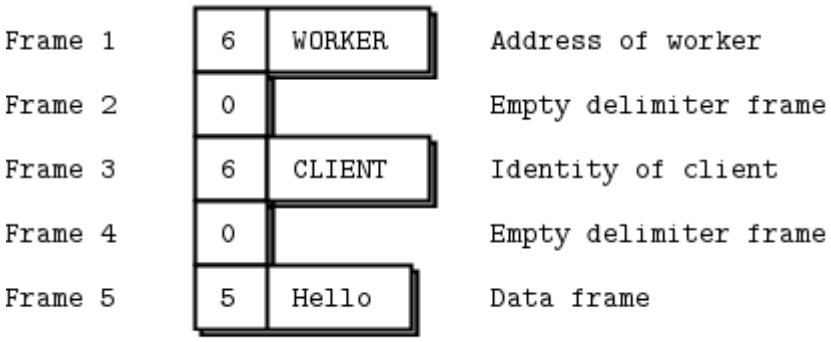
5) En el recorrido del mensaje, desde su envío por el socket REQ del cliente, y su paso por los 2 sockets ROUTER, se añaden varios segmentos o *frames*. Tal como se indica en la guía de ØMQ:

    a. Mensaje que envía el cliente (envía '*Hello*' como podría enviar '*work_harder*'):

    b. Mensaje que recibe el broker en su *frontend* (en nuestro caso, '*CLIENT*' será el identificador del cliente, pasado como argumento del mismo):



    c. Antes de reenviar el mensaje, el broker le añade dos segmentos más, y entonces lo envía mediante su *backend* (en nuestro caso, '*WORKER*' y '*CLIENT*' serán, respectivamente, los identificadores de trabajador y cliente). Hay que tener en cuenta que el primer segmento será usado por el *backend* para seleccionar al trabajador, y que el segundo segmento, vacío, es el delimitador que espera el socket REQ del trabajador:



6) El trabajador al que se le envíe la petición de servicio pasará al estado de 'no disponible'. Se mantendrá en ese estado hasta que envíe al broker una petición para que notifique al cliente que el servicio ha sido atendido.

7) En el camino de retorno (desde trabajador hasta cliente), se repite un proceso similar de transformación del mensaje:

a. El trabajador entrega su mensaje '*work_done*' con 3 segmentos: identificador de cliente, delimitador vacío, y segmento de datos.
b. El socket *backend* recibe un mensaje con 4 segmentos, los 3 que son propios del trabajador, y un delimitador adicional añadido por el socket REQ del trabajador. Dado que el *backend* es un socket ROUTER, añade el identificador del trabajador al mensaje que entrega al broker. Por tanto, el broker recibe un mensaje con 5 segmentos: identificador del trabajador, delimitador, identificador del cliente, delimitador, y el segmento de datos.
c. El bróker vuelve a añadir al trabajador a la lista de trabajadores disponibles, tras lo que retira esos 2 primeros segmentos, y envía, a través de su socket *frontend*, el mensaje con los otros 3 segmentos (identificador del cliente, delimitador, y segmento de datos).
d. El socket *frontend* retira el segmento del identificador del cliente, y entrega a ese cliente un mensaje con dos segmentos: delimitador, y el segmento de datos.
e. El socket REQ del cliente retira el segmento delimitador, y entrega sólo el segmento de datos (por ejemplo: '*work_done*').

8) Al igual que los trabajadores, el broker quedará en ejecución indefinida: siempre a la espera de recibir peticiones de servicio.

3.5 Implementación y primeras pruebas

Para probar el funcionamiento de nuestras primeras versiones de los programas *lbbroker.js*, *lbworker.js* y *lbclient.js*, seguiremos una estrategia prudente y conservadora: se pondrá en ejecución el proxy, sólo un trabajador, y sólo un cliente. En broker y trabajador se activará el modo *verbose*.

La finalidad de estas primeras pruebas es comprobar que los mensajes se transmiten adecuadamente entre cliente y trabajador, a través del broker, así como que se respeta lo relativo a transformaciones del mensaje a través de los sockets ROUTER, tal y como se ha explicado en el punto anterior.

El **modo *verbose*** permitirá obtener algo similar a una traza de la ejecución de los programas conforme se procesa una petición de servicio. En el Anexo 2 se muestra un ejemplo de la salida por consola de la ejecución de los programas en modo *verbose*. Este ejemplo se presenta como referencia: no para que vuestra implementación genere exactamente esa salida (cosa, por lo demás, imposible, dada la generación aleatoria de las identidades de clientes y trabajadores).

Para simplificar el código relativo al modo *verbose*, se proporciona (en el Anexo 1) una función llamada ***showArguments***, que debe invocarse allí donde un socket reciba un mensaje.

3.6 Más implementación y pruebas completas

Si se ha conseguido superar satisfactoriamente las pruebas del punto anterior, estamos en condiciones de incrementar el número de trabajadores y de clientes. Podremos entonces

verificar si nuestro broker reparte equitativamente el trabajo entre los integrantes de su *pool* de trabajadores.

Se recomienda hacer las primeras pruebas con un número moderado de trabajadores y clientes. Por ejemplo, con 3 trabajadores y 10 clientes. Sería conveniente modificar el broker de manera que lleve la cuenta del número de peticiones atendidas por cada trabajador, y que muestre (por consola) una tabla con los identificadores de los trabajadores y las peticiones atendidas por cada uno, una vez todos los clientes acaben.

Desde una misma terminal, podemos lanzar fácilmente la ejecución simultánea de, por ejemplo, dos trabajadores mediante:

```
bash-4.1$ node lbworker tcp://158.42.184.187:8002 e6ef6-67a38 Ready OK false &
          node lbworker tcp://158.42.184.187:8002 d34fe-4567c Ready OK false &
```

Sin embargo, hay que tener cuidado con la ejecución *batch*, o en segundo plano, de programas que no tienen una conclusión explícita. Sería recomendable modificar el *worker* de modo que tuviera una "vida limitada" (fijando un temporizador –función *setTimeout*– que lance el cierre del proceso: *process.exit();*).

Si se quiere hacer una prueba con un número significativamente grande de clientes, por ejemplo, 100 clientes, se debe recurrir a otra estrategia de ejecución simultánea de los programas: implementar un ***shell script*** [8].

[8] Una buena referencia al respecto es: http://linuxcommand.org/writing_shell_scripts.php. Donde se puede encontrar lo relativo a bucles: http://linuxcommand.org/wss0120.php#loops

Consideremos el siguiente *script* sencillo:

```
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 del esquema de este *script*, no debería ser difícil implementar un par de scripts, llamados ***lbclients_script.sh*** y ***lbworkers_script.sh***, que nos permitan, mediante una sola orden (la de su invocación), ejecutar un número bastante mayor de clientes y trabajadores, respectivamente. Los dos scripts nombrados forman parte de la entrega del proyecto ***lab2*** (deben ser añadidos a los depósitos al igual que los programas *js* ya citados).

4 APARTADO 3. MEJORAS Y VARIACIONES DEL PROXY CON COLAS ØMQ

En este apartado se realizarán modificaciones de los programas desarrollados en el punto anterior. Estas modificaciones se describen a continuación de una manera somera, es decir, no tan guiada como en los apartados ya completados.

Para ello, se ha de crear un nuevo directorio, llamado ***zmqexperto***[9]. Los programas a implementar en este proyecto se llamarán: ***lbbroker2.js***, ***lbworker2.js*** y ***lbclient2.js***. (Se tomará como punto de partida el código ya implementado en *lbbroker.js*, *lbworker.js* y *lbclient.js*).

Las modificaciones que se proponen son las siguientes (el orden de llevarlas a cabo no tiene por qué corresponder con el orden de su enumeración aquí):

- **Promises**: Introducir el uso de **promesas** en la gestión de las peticiones de los clientes al broker, y de las peticiones que el bróker realiza a cada trabajador. Comprobar si esta modificación tiene algún efecto sobre el comportamiento global del sistema de atención de peticiones de servicio.

- **Load Balancing**: Modificar, en el broker, el criterio de reparto equitativo de la carga entre los trabajadores por otro criterio basado en la carga efectiva que tengan éstos. Para ello, conviene recordar lo visto en la práctica 1 (función *getLoad* y programas que la usaban). Debéis encontrar una forma de hacer llegar esta información del trabajador al bróker.

- **Configuración dinámica**: El proxy o broker debe ser dinámicamente configurable, para lo que presentará un socket ØMQ de respuesta, que será usado desde un configurador que le enviará algún mensaje en formato JSON para determinar los criterios de *routing*.

[9] La conclusión de cada apartado de la práctica no implica que uno sea programador básico, avanzado y experto en ØMQ (algún nombre había que dar a los proyectos), pero ciertamente, si se completan los objetivos de la práctica, se podrá decir que se tiene una base sólida en el desarrollo de aplicaciones de Node.js con ØMQ.

4.1 Uso de promesas

Se desea introducir el uso de **promesas** en la gestión de las peticiones de un cliente al bróker, y de el bróker a un trabajador.

Como orientación de los cambios que hay que llevar a cabo, se presenta la siguiente variante del programa *hwclient.js* (ya estudiado, y disponible en el proyecto *zmq_nodejs_examples*), llamada ***hwclient2.js***, en el cual se usan promesas[10] para gestionar las respuestas a enviar a los clientes que se conecten a dicho servidor.

4.1.1 hwserver2.js

```
01:  // Hello World client in Node.js
02:  // Connects REQ socket to tcp://localhost:5559
03:  // Sends "Hello" to server
04:
05:  var zmq         = require('zmq')
06:    , makeSender  = require('auxfunctions').makeSender
07:    , requester   = zmq.socket('req');
08:    , makeRequest = makeSender(requester)
09:
10:  requester.connect('tcp://localhost:5559');
11:  var replyNbr = 0;
12:
13:  for (var i = 0; i < 10; ++i) {
14:    makeRequest("Hello").then(function(m) {
15:       console.log('got reply', replyNbr, msg.toString());
16:       replyNbr += 1;
17:    });
18:  }
```

Una vez implementada esta variante en nuestro sistema de atención de peticiones de servicio, deberemos comprobar, además de su correcto funcionamiento, si la modificación tiene algún efecto sobre el comportamiento global del sistema. Estudiar el funcionamiento de la función "makeSender" introducida. En particular, observar las condiciones bajo las cuales funciona correctamente.

4.2 Equilibrio de carga

Se desea modificar el criterio de equilibrio de carga (*load balancing*) usado en el proyecto anterior. Queremos que el broker no use el criterio de reparto equitativo de la carga entre los trabajadores, sino otro criterio, basado en la carga efectiva que tengan éstos.

Recordemos que se dispone de una función que devuelve la carga de trabajo de una máquina dada, llamada ***getLoad***, cuyo código se incluye en el módulo *auxfunctions.js* (véase el Anexo 1).

Además, conviene leer el punto 4.2 de la práctica 1 (el Subapartado B de la Aplicación final: Proxy TCP/IP inverso). La idea será la siguiente:

[10] En el caso de que el programa *hwclient2.js* no funcione, generando el error "***Cannot find module 'bluebird'***", se deberá a que no se tiene instalado correctamente el módulo bluebird. La solución es instalarlo, desde el $HOME del usuario, ejecutando: "**npm install bluebird**".

- Los trabajadores (servidores) incluirán su carga efectiva en todo mensaje que envíen al bróker. El bróker tomará nota de esta carga para implantar su algoritmo de balanceo de carga.

- Para cada trabajador registrado en el bróker, éste enviará una petición nula al trabajador pasado un tiempo configurable. Un trabajador recibiendo una petición nula simplemente responde (con su nueva información de carga). El bróker usa la nueva información de carga incluida en la respuesta.

Una vez implementada esta nueva variante en nuestro sistema de atención de peticiones de servicio, deberemos comprobar su correcto funcionamiento, es decir, si se aplica el criterio de *routing* de enviar las peticiones al trabajador que, en cada momento, tenga la carga menor.

4.3 Configuración dinámica

Se quiere que el broker (proxy) disponga de un socket ØMQ de respuesta, que estará a disposición de un nuevo programa, el configurador (***lbconfig2.js***).

Este configurador usará un socket ØMQ de petición para enviar un mensaje en formato JSON con la especificación de nuevos criterios de routing. En cuanto el broker lea una petición de su configurador, aplicará el nuevo criterio de routing a las peticiones de servicio que le vayan llegando.

Los mensajes de petición del programa configurador, *lbconfig2.js*, deberían ajustarse a uno de los siguientes formatos (según el criterio que se desee aplicar):

```
var msg1 = JSON.stringify({
    "distribution":"equitable",
    "adjustFactor":number
});
```

```
var msg2 = JSON.stringify({
    "distribution":"lowerLoad",
    "periodicity":secs,
    "lowLoadWorkers":number
});
```

El primer mensaje (*msg1*) corresponde a la activación del criterio de reparto equitativo de las peticiones (especificado en el mensaje como *"distribution":"equitable"*). Se ha de proporcionar, además, un número real (especificado en el mensaje como *"adjustFactor":number*), en el rango [0.1 - 0.9], que usará el broker como factor corrector en las desviaciones que pudieran existir respecto a un exacto reparto equitativo. El broker usará ese factor para "gratificar" al trabajador o trabajadores que hayan trabajado más (reduciendo, de algún modo, la probabilidad de ser seleccionados), y "exigir más" al trabajador o trabajadores que hayan atendido menos peticiones de los clientes (incrementando la probabilidad de ser seleccionados).

El segundo mensaje (*msg2*) corresponde a la activación del criterio de reparto teniendo en cuenta la carga real de los trabajadores (especificado en el mensaje como *"distribution":"lowerLoad"*). Además, se han de proporcionar otros dos datos: un número entero (especificado en el mensaje como *"periodicity":secs*), que usará el broker como nuevo valor de los segundos para el temporizador que envía las peticiones de consulta de carga a los trabajadores; y un número entero (especificado en el mensaje como *"lowLoadWorkers": number*), que usará el broker para asignar una nueva petición entre varios trabajadores con poca carga (si ese número es 3, el broker seleccionará aleatoriamente un trabajador del subconjunto de los 3 trabajadores con menos carga; si ese número es 2, lo elegirá del subconjunto de los 2 con menos carga; si ese número es 1 no podrá elegir, seleccionará obligatoriamente el trabajador con la carga menor).

5 ENTREGA

Jueves	15-Enero

En la fecha de entrega, conjunta con la práctica 3, el depósito Git canónico de cada equipo debe contener todo el material necesario:

1. Copia de la memoria de la práctica, incluyendo identificación de los alumnos del equipo y el código fuente comentado.
2. Ficheros con cada una de las aplicaciones entregables.

5.1 Materiales a incluir

El **depósito Git** canónico, **lab2**, en su master branch deberá contar con los siguientes directorios:

1. **zmqbasico** (hwserver.js, hwclient.js, publisher.js, subscriber.js, rrbroker.js, rrworker.js, rrclient.js).
2. **zmqavanzado** (lbbroker.js, lbworker.js, lbclient.js, auxfunctions.js, lbworkers_script.sh, lbclients_script.sh)
3. **zmqexperto** (lbbroker2.js, lbworker2.js, lbclient2.js, lbconfig2.js)
4. **zmqmemoria** (memoriaTSR2)

El contenido de la **memoria de la práctica** (memoriaTSR2 en formato PDF, ODT o DOC) debe seguir la siguiente estructura:

- Portada:
  - Grupo de laboratorio
  - Relación de integrantes del trabajo (apellidos, nombre, correo)
  - Tabla de contenidos
- Repetir para cada proyecto:
  - Nombre y muy breve descripción de cada proyecto
  - Problemas relevantes y su solución
    - Líneas relevantes de código
- Detalles específicos de algunos proyectos (incluir en el apartado anterior):
  - Cosas que se han destacado a lo largo de la práctica con preguntas específicas
- Comentarios globales
- Bibliografía consultada
- Anexo con los fuentes (misma organización que la mostrada acerca del depósito)

6 ANEXO 1. FUNCIONES AUXILIARES

6.1.1 auxfunctions.js

```
01:    module.exports = {
02:
03:       // *** makeSender function
04:
05:       makeSender : function (reqsock) {
06:         var Promise = require('bluebird');
07:         var promises = [];
08:
09:         reqsock.on('message', function (reply) {
10:            promises.shift().resolve(reply);
11:         });
12:
13:         reqsock.on('error', function(reason) {
14:            promises.shift().reject(reason);
15:         });
16:
17:         return function (message) {
18:            return new Promise(function (resolve, reject) {
19:                promises.push({resolve: resolve, reject: reject});
20:                reqsock.send(message);
21:            })
22:         };
23:       }
24:
25:       // *** getLoad function
26:
27:       getLoad : function() {
28:          var fs     = require('fs')
29:            , data   = fs.readFileSync("/proc/loadavg") // version sincrona
30:            , tokens = data.toString().split(' ')
31:            , min1   = parseFloat(tokens[0])+0.01
32:            , min5   = parseFloat(tokens[1])+0.01
33:            , min15  = parseFloat(tokens[2])+0.01
34:            , m      = min1*10 + min5*2 + min15;
35:          return m;
36:       },
37:
38:      // *** randNumber function
39:
40:      randNumber : function(upper, extra) {
41:         var num = Math.abs(Math.round(Math.random() * upper));
42:         return num + (extra || 0);
43:      },
44:
45:      // *** randTime function
46:
47:      randTime : function(n) {
48:         return Math.abs(Math.round(Math.random() * n)) + 1;
49:      },
50:
51:      // *** showArguments function
52:
53:      showArguments : function(a) {
54:         for (var k in a)
55:            console.log('\tPart', k, ':', a[k].toString());
56:      }
```

```
57:     }
```

6.1.2 randString.js

```
01:     function randString () {
02:       var len     = 10
03:         , charSet = '0123456789abcdef'
04:         , result  = [];
05:
06:       for (var i = 0; i < len; ++i) {
07:          result.push(charSet[Math.floor(Math.random() * charSet.length)]);
08:       }
09:       result.splice(len / 2, 0, ['-']);
10:       return result.join('');
11:     };
12:
13:     console.log( randString() );
```

7 ANEXO 2. EL MODO VERBOSE

7.1.1 Ejecución de lbbroker.js en modo verbose

```
bash-4.1$ node lbbroker 8001 8002 true
broker: frontend-router listening on tcp://*:8001 ...
broker: backend-router listening on tcp://*:8002 ...
received request: "I_am_ready" from worker ( e6ef6-67a38 )
       Part 0 : e6ef6-67a38
       Part 1 :
       Part 2 : I_am_ready
received request: "work_harder" from client ( 7f2ba-2376e )
       Part 0 : 7f2ba-2376e
       Part 1 :
       Part 2 : work_harder
sending client ( 7f2ba-2376e ) req to worker ( e6ef6-67a38 ) through bakend
       Part 0 : e6ef6-67a38
       Part 1 :
       Part 2 : 7f2ba-2376e
       Part 3 :
       Part 4 : work_harder
received request: "OK" from worker ( e6ef6-67a38 )
       Part 0 : e6ef6-67a38
       Part 1 :
       Part 2 : 7f2ba-2376e
       Part 3 :
       Part 4 : OK
sending worker ( e6ef6-67a38 ) rep to client ( 7f2ba-2376e ) through frontend
       Part 0 : 7f2ba-2376e
       Part 1 :
       Part 2 : OK
```

7.1.2 Ejecución de lbworker.js en modo verbose

```
bash-4.1$ node lbworker tcp://158.42.184.187:8002 e6ef6-67a38 I_am_ready OK
true
worker ( e6ef6-67a38 ) connected to tcp://158.42.184.187:8002 ...
worker ( e6ef6-67a38 ) has sent READY msg: "I_am_ready"
worker ( e6ef6-67a38 ) has received request: "work_harder" from client ( 7f2ba-2376e
)
       Part 0 : 7f2ba-2376e
       Part 1 :
       Part 2 : work_harder
worker ( e6ef6-67a38 ) has send its reply
       Part 0 : 7f2ba-2376e
       Part 1 :
       Part 2 : OK
worker ( e6ef6-67a38 ) has sent 1 replies
```

7.1.3 Ejecución de lbclient.js

```
bash-4.1$ node lbclient tcp://158.42.184.187:8001 7f2ba-2376e work_harder
client ( 7f2ba-2376e ) connected to tcp://158.42.184.187:8001 ...
client ( 7f2ba-2376e ) has sent its msg: "work_harder"
client ( 7f2ba-2376e ) has received reply: "OK"
```