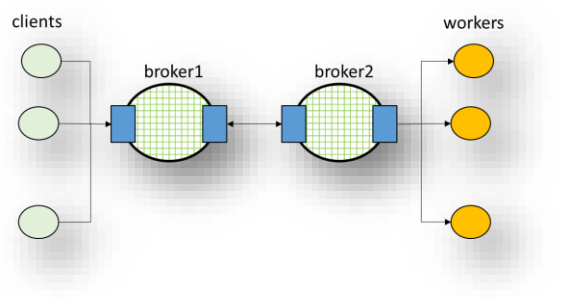
TSR – SOLUCIONES EXAMEN LABORATORIO 2, 02-12-2021

Pregunta 1 (4 puntos) Los dos programas siguientes (**broker1** y **broker2**) representan uno de los posibles intentos de solución al problema de dividir el broker en dos, tal y como se propone en la práctica 2

broker1.js

```
01: const zmq = require('zeromq')
02: let nw=0, cli=[], msg=[]
03: let sc = zmq.socket('router')
04: let sb = zmq.socket(*****)
05: sc.bind('tcp://*:9990')
06: sb.bind('tcp://*:9991')
07:
08: function dispatch(c,m) {
09:    nw--
10:    sb.send([c,'',m])
11: }
12:
13: sc.on('message',(c,sep,m) => {
14:    if (nw!=0) dispatch(c,m)
15:    else {cli.push(c); msg.push(m)}
16: })
17:
18: sb.on('message',(c,sep,r) => {
19:    nw++
20:    if (c!='') sc.send([c,'',r])
21:    //
22: })
```

NOTA.- asume que clientes y workers son parametrizables con su id y url donde conectar (por lo demás son los mismos usados en prácticas)



broker2.js

```
01: const zmq = require('zeromq')
02: let workers=[]
03: let sb = zmq.socket(*****)
04: let sw = zmq.socket('router')
05: sb.connect('tcp://localhost:9991')
06: sw.bind('tcp://*:9992')
07:
08: sw.on('message', (w,sep,c,sep2,r) => {
09:    workers.push(w)
10:    sb.send([c,'',r])
11:
12: })
13:
14: sb.on('message', (c,sep,m) => {
15:   sw.send([workers.shift(),'',c,'',m])
16: })
```

Responde de forma razonada a las siguientes cuestiones:

a) Indica el tipo de socket que podemos usar para el socket `sb` de broker1 y para el socket `sb` de broker2 (líneas 4 de broker1 y 3 de broker2). Justifica la respuesta.

Solución:

Las características necesarias en los sockets entre brokers deben permitir la comunicación asincrónica y bidireccional. La asincronía permitirá la atención de nuevas peticiones sin necesidad de esperar la conclusión de las pendientes, y la bidireccionalidad permite circular tanto peticiones (de izquierda a derecha) como respuestas (en la dirección opuesta).

Partiendo de cero podríamos seleccionar `dealer` para ambos (2 sockets), o bien un par `push/pull` también para ambos (4 sockets). Sería más extraño el uso de sockets `router`, pero viable. En el código del enunciado sólo es factible la primera opción.

En el broker1:

```
let sb = zmq.socket('dealer')
```

Y en broker2:

```
let sb = zmq.socket('dealer')
```

b) Justifica si el código propuesto funcionará correctamente en aquellas situaciones relacionadas con la disponibilidad de workers.

Solución:

La disponibilidad de workers se encuentra reflejada en la cola `workers` de broker2.

- Cada vez que se añade un worker (línea 9 de broker2), se envía un mensaje a broker1 (línea 10), provocando el incremento del contador `nw` en broker1.
- Cada vez que se retira un worker (línea 15 de broker2), se debe a la llegada de un mensaje enviado (`sb.send`) por broker1. Esto únicamente ocurre en la función `dispatch`, que decrementa el contador `nw` en broker1.

En resumen, `nw` se incrementa al añadir un worker disponible, y se decrementa al retirarlo, por lo que `nw` siempre representa la longitud de la cola `workers`.

La condición de disponibilidad de workers se evalúa a la llegada de peticiones de clientes (*listener* en las líneas 13 a 16 de broker1), y debería comportarse así:

- `Si hay workers disponibles: pasar la petición a broker2 (la petición puede ser atendidas)`
- `Si no los hay: encolar la petición en broker1`

Y eso es justamente lo que hace. Por tanto la disponibilidad de workers está adecuadamente representada y gestionada.

c) Justifica si debería añadirse código adicional a partir de la línea 21 de broker1 para que el doble broker tuviera un funcionamiento correcto. De ser así, escribe ese código.

Solución:

Este caso presenta cierta analogía con el anterior, porque se trata de la gestión de una cola (de clientes) que no ha considerado todas las eventualidades. Concretamente ha quedado sin cubrir la posibilidad de que un worker quede libre (tras devolver una respuesta) mientras existen peticiones de clientes en espera.

El lugar en el código en que esto puede ser detectado se encuentra en el *listener* de las líneas 18 a 22 de broker1, donde deberá agregarse (línea 21):

```
if (cli.length!=0) dispatch(cli.shift(),msg.shift())
```

Pregunta 2 (1 punto) La versión del **broker con tolerancia a fallos** incluida en la práctica 2 permite experimentar con ciertos escenarios de error. Debes contestar a las siguientes preguntas relacionadas con ese sistema:

a) Explica qué ocurre si el broker falla. Debes mencionar el efecto sobre el resto de componentes, las posibles peticiones en curso y el sistema completo.

Solución:

Si el broker falla ya no hay forma de que el resto de componentes pueda comunicarse: no se atienden peticiones ni se devuelven respuestas a las ya admitidas. El sistema deja de funcionar por completo.

b) Explica qué ocurre si falla un worker. Debes incluir las diferencias entre los casos en los que, en el momento del fallo, el worker estuviera procesando una solicitud o a la espera.

Solución:

Si un worker en situación disponible (sin trabajo asignado) falla, aparecerá como una entrada en la cola `workers`. En un momento dado el broker puede encargarle un trabajo que nunca será recibido, provocando que venza el temporizador iniciado. Como consecuencia será marcado como erróneo (en `failed[]`), y una copia de ese trabajo será encargada a otro worker.

En el caso en que un worker falle mientras procesa una petición, las consecuencias serán las mismas porque el temporizador iniciado en el broker vencerá.

c) Explica qué ocurre si un worker, que el broker consideró *averiado*, devuelve con retraso al broker una respuesta `r` a la solicitud `m` de un cliente c.

Solución

Los workers *averiados* se representan con el objeto `failed[]`, actuando como una *"lista negra"*, de manera que el broker descarta los mensajes procedentes de los workers señalados por dicha lista. De esa manera las solicitudes de los clientes no pueden crear respuestas duplicadas.

Pregunta 3 (4 puntos) Disponemos de un **sistema de chat** idéntico al descrito en el apartado 5 de la práctica 2, del que se muestra el código de un cliente.

```
01: const zmq = require('zeromq')
02: const nick='Ana' //Assume it's random.
03: let sub = zmq.socket('sub')
04: let psh = zmq.socket('push')
05: sub.connect('tcp://127.0.0.1:9998')
06: psh.connect('tcp://127.0.0.1:9999')
07: sub.subscribe('')
08: sub.on('message', (nick,m) => {
09:    console.log('['+nick+']'+m)
10: })
11: process.stdin.resume()
12: process.stdin.setEncoding('utf8')
```

```
13: process.stdin.on('data'  ,(str) => {
14:    psh.send([nick, str.slice(0,-1)])
15: })
16: process.stdin.on('end',() => {
17:    psh.send([nick, 'BYE'])
18:    sub.close(); psh.close()
19: })
20: process.on('SIGINT',() => {
21:    process.stdin.end()
22: })
23: psh.send([nick,'HI'])
```

Uno de los participantes en dicho chat (el "*mafioso*") ha ideado una forma de abusar del sistema por medio de clientes del chat ficticios ("*esbirros*") que obedecen sus órdenes. Su operativa es…

- Cuando el mafioso lee un mensaje del chat (*mensaje_original*) procedente de un cliente concreto (lo denominamos *objetivo*), reaccionará ordenando que cada *esbirro* envíe un mensaje al chat con contenido "**No me gusta el mensaje de *objetivo*: *mensaje_original***".
- Tanto el *mafioso* como los *esbirros* serán versiones modificadas del cliente genérico de chat. Es extremadamente conveniente conocer que **su implementación solo añade instrucciones al original**, manteniendo una separación limpia entre la parte *cliente de chat* y la parte de interacción *mafioso-esbirros*.
    - La única excepción es el momento que provoca la reacción del *mafioso*.
- El *mafioso* y los *esbirros* reciben desde la línea de órdenes, en su variable ***port***, el número de puerto a utilizar para intercomunicarse. El mafioso recibe también en su variable

target el identificador del usuario a molestar. No es necesario escribir código para asignar valor a esas variables en las cuestiones c) y d).

Diseña el código de mafioso y esbirros para contestar a las siguientes cuestiones:

a) Elige el/los tipo/s de socket/s ZeroMQ para comunicar mafioso y esbirros. Argumenta la elección.

Solución:

Puesto que la comunicación es iniciada por el mafioso y dirigida a todos los esbirros para transmitir la misma orden, parece adecuado un esquema con publicador (el mafioso) y suscriptores (los esbirros). Además no se necesita bidireccionalidad puesto que los esbirros no envían al publicador ni se comunican entre sí.

Para el *mafioso*, el código relevante sería:

```
let so = zmq.socket('pub')
so.bind('tcp://*:'+port)
```

Y para los *esbirros*:

```
let so = zmq.socket('sub')
so.subscribe('')
so.connect('tcp://127.0.0.1:'+port) // suponiendo que se encuentran en la misma máquina
```

b) El código del mafioso se basa en el cliente de chat. ¿Qué instrucciones insertarías en ese código para detectar el mensaje del objetivo e iniciar la reacción?. Indica dónde referenciando el/os números de línea del código del cliente de chat.

Solución:

Dentro del listener entre las líneas 8 y 10, se añadiría

```
if (nick==target) so.send('No me gusta el mensaje de '+nick+': '+m)
```

Provocando que una condición detectada en los mensajes del chat desencadene una orden en el canal del socket que comunica con los esbirros.

c) Siguiendo con el *mafioso*, escribe las instrucciones que añadirías al código del cliente de chat para construir el programa `mafioso`.

Solución:

Todas las instrucciones necesarias ya han sido mencionadas en los dos apartados anteriores. No se necesita nada más.

d) Centrándonos en los *esbirros*, escribe las instrucciones que añadirías al código del cliente de chat para construir el programa `esbirro`.

Solución:

Tras la última línea de ese cliente de chat, junto a las órdenes relacionadas con el apartado a), añadimos:

```
so.on('message', (m) => psh.send([nick,m]))
```

Pregunta 4 (1 punto) En uno de los apartados de la práctica 2 se pide visualizar periódicamente **estadísticas sobre las peticiones atendidas**, pero en esta pregunta no nos interesa el número total. Explica qué estructuras de datos has diseñado para mantener la información necesaria **para cada worker**, y cómo accedes a las mismas. Ilústralo implementando la función que, según el enunciado, debería mostrar esa información cada 5 segundos en pantalla (por ejemplo, una función `visualize()` invocada mediante `setInterval(visualize, 5000)`)

Solución:

Esta pregunta solicita explicar algunos detalles de la **solución que el alumno o la alumna debió desarrollar** para una parte específica de la práctica.

Necesitamos una estructura de datos, indexada por identificador de worker (su cola), que contenga un contador para cada uno. Dado que no conocemos de antemano la cantidad de workers, no podemos dimensionar un vector de la forma clásica. Afortunadamente JavaScript es muy flexible para emplear vectores como diccionarios, y recorrerlos sin que el índice sea un entero (hay un uso análogo en el objeto `tout` de `ftbroker`).

Por ejemplo:

```
let counters = {} // declaration
```

Dado un worker w que acaba de finalizar un trabajo, el incremento de su contador será:

```
counters[w]++
```

La inicialización es un caso especial que se integra con el general, de forma que la anterior instrucción se convierte en un `if`:

```
if (counters[w]) counters[w]++
else counters[w]=1
```

Para el caso concreto de la función `visualize` invocada para mostrar estos contadores, una posible implementación sería:

```
function visualize(){
  for (let w in counters) console.log("worker "+w+" completed "+counters[w]+" jobs")
}
```