Práctica 3: Bróker para NodeJS mediante ØMQ

Tecnologías de los Sistemas de Información en la Red.

Introducción

En clase de teoría se han presentado los sockets router y dealer y se ha mostrado un primer bróker realizado mediante estos sockets. Debes estudiarlo y tenerlo presente al realizar esta práctica.

En esta práctica emplearemos dos sockets de tipo router en el bróker. Por tanto, tanto clientes como trabajadores emplearán sockets de tipo REQ. Nótese que los trabajadores ya no serán de tipo REP, lo que implicará algunos cambios en el código y en el funcionamiento general del bróker.

Objetivos

- Asentar los conceptos teóricos introducidos en el tema 3
- Profundizar en el patrón bróker (proxy inverso)
- Introducir algunas ideas prácticas para tolerancia a fallos
- Reforzar los conceptos teóricos practicando programación asincrónica para sistemas distribuidos.

Propuesta para la organización del tiempo

- Sesión 1.- **Bróker router-router**: pruebas del bróker y desarrollo de 2 modificaciones.
- Sesión 2.- **Bróker tolerante a fallos**. Pruebas del bróker.

Método de trabajo

- Por simplicidad, lanzamos los distintos componentes de cada aplicación en la misma máquina (IP = localhost)
    - Pero también se pueden repartir los componentes sobre máquinas distintas
- Los programas requieren argumentos en línea de órdenes
- En las distintas tareas se plantean cuestiones que el alumno debe responder
- El fichero `fuentes3.zip` contiene el código para realizar las distintas pruebas y la base para hacer nuestras modificaciones.

Biblioteca tsr.js

El código de los bróker que trabajamos sigue empleando la biblioteca “tsr.js”. La incluimos a continuación como referencia.

```
const zmq  = require('zeromq')
const path = require('path')
module.exports = {zmq, error, lineaOrdenes, traza, adios, creaPuntoConexion, conecta}

function error(msg) {console.error(`\x1b[31m ${msg} \x1b[0m `); process.exit(1)}

function lineaOrdenes(params) { //Comprobacion de parametros en linea de órdenes. Crea variables
asociadas
    var args = params.split(" "), prog = path.basename(process.argv[1])
    if( process.argv.length != (args.length+2) )
        error(`Parámetros incorrectos. Uso: node ${prog} ${names}`)
    args.forEach((param,i) => {global[param] = process.argv[2+i]})
    console.log(`Arranca el programa ${prog} con los siguientes parámetros en línea de
órdenes:`)
    for (let a in args) {console.log(`\t${args[a]}\t|${global[args[a]]}|`)}
}
function traza(f,names,value) { //muestra los argumentos al invocar la función f
    console.log(`funcion ${f}`)
    var args = names.split(" ")
    for (let a in args) console.log(`\t${args[a]}\t|${value[a]}|`)
}
function adios(sockets, despedida) {        //cierra la conexión y el proceso
    return ()=>{
        console.log(`\x1b[33m ${despedida} \x1b[0m`)
        for (let s in sockets) sockets[s].close()
        process.exit(0)
    }
}
function creaPuntoConexion(socket,port) {
    console.log(`Creando punto conexión en port ${port} ...`)
    socket.bind(`tcp://*:${port}`, (err)=>{
        if (err) error(err)
        else console.log(`Escuchando en el port ${port}`)
    })
}
function conecta(socket,ip,port) {
    console.log(`Conectando con ip ${ip} port ${port}`)
    socket.connect(`tcp://${ip}:${port}`)
}
```

SESIÓN 1.- Bróker router-router: prueba y modificaciones

Trabajaremos con el código fuente incluido en la carpeta “broker-workers”. Ahí puedes encontrar estos ficheros de código:

- **`brokerRepReq.js`** : Bróker que emplea el socket REP como frontend y socket REQ para el backend.
- **`brokerRouterDealer.js`**: Bróker que emplea el socket ROUTER para el frontend y socket DEALER para el backend
- **`brokerRouterRouter.js`**: Bróker que emplea sockets ROUTER para frontend y backend.
- **`cliente.js`**: Cliente REQ que funciona con todos los bróker. Indicaremos como argumentos, la identidad del cliente, cuántas peticiones realizar, y el host y puerto donde conectar (frontend)
- **`workerRep.js`**: Trabajador que emplea socket REP. Necesita como argumentos la identidad del worker y host y puerto donde conectar (backend)
- **`workerReq.js`**: Trabajador que emplea socket REQ. Necesita como argumentos la identidad del worker y host y puerto donde conectar (backend)
- Para todos los bróker, usaremos como argumentos, primero el puerto para el frontend (los clientes) y el puerto para el backend (los trabajadores)

En esta sesión haremos las siguientes tareas:

1. Bróker router/dealer y bróker rep/req: Estudio del código y pruebas
2. Bróker router/router: Estudio del código y pruebas
3. Estadísticas del bróker: Modificación del bróker router/router para que ofrezca estadísticas
4. Bróker doble: Implementación de un bróker doble, un bróker para clientes y un bróker para trabajadores.

1.1.- Bróker router/dealer y bróker rep/req (10 minutos)

Tienes disponible como inicio a esta práctica la implementación del bróker router/dealer, similar a la implementación vista en clases de teoría. También tenemos la implementación de un bróker rep/req

Pruebas a realizar (como mínimo: deberías tratar de hacer más pruebas):

- terminal 1) `node brokerRouterDealer.js  8888 8889`
- terminal 2) `node workerRep.js W1 localhost 8889`
- terminal 3) `node workerRep.js W2 localhost 8889`
- terminal 4) `node cliente.js C1 3 localhost 8888`
- terminal 5) `node cliente.js C2 3 localhost 8888`

Preguntas:

1. El socket dealer envía las peticiones mediante round-robin. Razona qué limitaciones ofrece esta aproximación para un bróker.
2. Razona si los procesos deben lanzarse en determinado orden.
3. Indique qué ocurre si durante la ejecución, matamos uno de los trabajadores.
4. Prueba a lanzar muchos clientes. Los puedes lanzar desde el mismo terminal, en una única línea, uniendo las órdenes mediante el operador de segundo plano '&'. Por ejemplo:
   1. `node cliente.js … & node cliente.js … & node cliente.js...`
5. Observa qué trabajador se encarga de cada una de las peticiones efectuadas por los clientes.

1.Si tenemos más de un worker las peticiones se envían a ambos utilizando Round Robin, sin tener en cuenta la diferencia en potencia o carga de trabajo que tengan los ordenadores worker

2. No importa el orden gracias al la asincronidad

3. Si matamos un trabajador, la petición del cliente quedará desatendida y este quedará suspendido esperando una respuesta. El broker no quedará bloqueado, en cambio. Este continuará procesando respuestas y peticiones de otros workers y clientes.

5. Debido al Round Robin cada worker atenderá a la mitad de las peticiones realizadas (RR distribuye de manera equitativa). El orden de recepción, en cambio, no sigue una secuencia concreta: en algunas ejecuciones, un worker podrá realizar las respuestas de los 3 mensajes de un mismo cliente.

Sustituye el bróker router/dealer por el bróker rep/req. Para ello repite las pruebas que has realizado, lanzando ahora el bróker rep/req y lanza los clientes en una sola línea enlazando las órdenes con el operador de segundo plano '&:

- terminal 1) `node brokerRepReq.js  8888 8889`
- terminal 2) `node workerRep.js W1 localhost 8889`
- terminal 3) `node workerRep.js W2 localhost 8889`
- terminal 4) `node cliente.js C1 3 localhost 8888 & node cliente.js C2 3 localhost 8888`

Preguntas:

¿Qué diferencia de funcionamiento y rendimiento observas entre el bróker router/dealer y el bróker rep/req?

Funcionamiento: con el broker rep/req las peticiones y respuestas son atendidas de manera secuencial, lo que causa que el broker solo atenderá al siguiente mensaje del cliente 2 después de devolver la respuesta del mensaje del cliente 1.

Rendimiento: este broker se queda bloqueado al enviar el trabajo a los trabajadores. En caso de que el worker cayera, el broker dejaría de funcionar. Esto no ocurría con el anterior broker

1.2.- Bróker router/router (10 minutos)

Los bróker vistos en la sección anterior presentan limitaciones. Debes entender qué limitaciones tienen y qué mejoras obtendremos con un bróker router/router

brokerRouterRouter.js

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
lineaOrdenes("frontendPort backendPort")

let workers  =[] // workers disponibles
let pendiente=[] // peticiones no enviadas a ningun worker
let frontend = zmq.socket('router')
let backend  = zmq.socket('router')


creaPuntoConexion(frontend, frontendPort)
creaPuntoConexion(backend,  backendPort)

function procesaPeticion(cliente,sep,msg) => { // llega peticion desde cliente
   traza('procesaPeticion','cliente sep msg',[cliente,sep,msg])
   if (workers.length) backend.send([workers.shift(),'',cliente,msg])
   else pendiente.push([cliente,msg])
}


function procesaMsgWorker(worker,sep,cliente,resp) => {
    traza('procesaMsgWorker','worker sep cliente resp',[worker,sep,cliente,resp])
    if (pendiente.length) { // hay trabajos pendientes. Le pasamos el mas antiguo al worker
        let [c,m] = pendiente.shift()
        backend.send([worker,'',c,m])
    }
    else workers.push(worker) // añadimos al worker como disponible
    if (cliente) frontend.send([cliente,'',resp]) // habia un cliente esperando esa respuesta
}

frontend.on('message', procesaPeticion)
frontend.on('error'  , (msg) => {error(`${msg}`)})
backend.on('message', procesaMsgWorker)
backend.on('error'  , (msg) => {error(`${msg}`)})
process.on('SIGINT' , adios([frontend, backend],"abortado con CTRL-C"))
```

Repite las pruebas efectuadas en la sección anterior con este nuevo bróker. Lógicamente en las pruebas debes emplear el trabajador de tipo “REQ” y no el de tipo “REP”.

Preguntas:

1. Qué ventajas ofrece este bróker respecto a los 2 anteriores ---------- ?
2. Qué sucede si nos equivocamos y lanzamos un trabajador de tipo “REP” en lugar de un trabajador de tipo REQ empleando este bróker. ¿Qué observas durante esta ejecución?

2. El funcionamiento del programa no es correcto. El router para poder enviar a un mensaje a un worker necesita que el worker le haya contactado primero, tal que el router pueda asignarle a ese canal un identificador. Al utilizar un REP socket, el worker no enviará ningún mensaje sin recibir antes uno request. Debido a esto, el router no podrá comunicarse con el worker

1.3.- Estadísticas bróker (20 minutos)

Modifica el bróker implementado en `brokerRouterRouter.js` para que mantenga el total de peticiones atendidas y el número de peticiones atendidas por cada trabajador.

- El bróker debe mostrar dicha información en pantalla cada 5 segundos.

Preguntas:

1. Indica la estrategia que has empleado (qué estructuras de datos has empleado y cómo actualizas estas estructuras de datos) para mantener en el bróker estadísticas separadas para cada worker.
2. Si llega una petición y se la pasamos al worker W, ¿debemos incrementar el número de peticiones atendidas por W (y el total) en ese momento, o cuando llegue la respuesta desde W?

1. Utilizando un array de tamaño n, siendo n un valor lo suficientemente grande para poder incluir a los workers necesarios.

Otra opción más costosa espacialmente pero también más eficiente y útil sería utilizando maps, tales que mapeen el nombre de un worker a un valor o contador. Así pues, cuando recibimos una respuesta de un worker, mapearíamos su counter y lo incrementaríamos.

2. Deberíamos incrementar el contador de peticiones atendidas por un worker W cuando recibamos la respuesta de este, ya que no podemos contar que un worker atiende una petición si en medio del proceso este worker cae.

1.4.- Bróker doble: bróker para clientes + bróker para workers (60 minutos)

Suponga un escenario donde queremos que un bróker se encuentre cerca de los clientes y otro bróker se encuentre cerca de los trabajadores. Entre estos 2 bróker tendremos una conexión privada de alta velocidad. Además queremos que no sea necesario modificar los clientes y los trabajadores.

- Tomamos como punto de partida el bróker implementado con brokerRouterRouter.
- Se trata de reescribir el código para tener **dos bróker** interconectados entre sí a los que llamamos broker1 y broker2. Crearemos 2 ficheros de código que llamaremos broker1.js y broker2.js.
- broker1 conoce a los clientes, y mantiene la cola de peticiones pendientes. Por tanto los clientes sólo interactúan con el broker1.
- broker2 conoce a los workers, y se responsabiliza de mantener los workers disponibles y repartir la carga entre ellos. Por tanto los workers sólo interactúan con el broker2.
- Todo cliente envía su petición a broker1, que la guarda en la cola de peticiones pendientes o la pasa a broker2, si éste puede atenderla.
- broker1 no sabe qué workers están disponibles, pero puede saber cuántos hay disponibles o simplemente si hay alguno disponible.
- La solución elegida debe mantener las mismas características externas (o sea, frente a clientes y frente a workers) que el código original. Por tanto los clientes y los trabajadores no deben modificarse.
- Cuando finalmente el broker1 reenvía la petición de un cliente a broker2, éste la envía al worker correspondiente.
- La respuesta del worker llega a broker2, que la pasa a broker1, y éste al cliente.
- El alta de un worker llega a broker2, y si se considera necesario, puede informar a broker1, mediante un contador de número de trabajadores libres, por ejemplo.
- Debes decidir cómo se comunican broker1 y broker2 (hay más de una alternativa) Mediante qué sockets y qué mensajes intercambian.
- Suponemos únicamente un proceso broker1 y un proceso broker2. No se pide que se soporte disponer de más procesos bróker.

Preguntas:

1. ¿Mediante qué par de sockets has conectado a los 2 bróker?
2. Razone qué otros tipos de sockets podrías haber empleado y cuáles no sería buena idea emplear para la conexión entre broker1 y broker2.
3. ¿Qué mensajes envía broker2 a broker1, además de las respuestas a las peticiones de los clientes?

SESIÓN 2.- Bróker tolerante a fallos

Durante esta sesión debes terminar el trabajo empezado en la sesión anterior y completar las tareas que se proponen para esta sesión.

En sistemas distribuidos la tolerancia a fallos es crucial. Todo sistema debe tratar los casos de fallo y continuar ofreciendo servicio incluso si alguna componente falla.

Durante esta sesión de prácticas dotaremos de tolerancia a fallos a los trabajadores. Si alguno de ellos falla, el trabajo será realizado por algún trabajador que quede disponible.

2.1.- Fallos sin emplear el bróker tolerante a fallos (20 minutos)

Como primer paso, veamos el comportamiento del bróker convencional visto hasta ahora "brokerRouterRouter", en caso de que falle algún worker.

Nos situamos en el directorio broker-worker y lanzamos:

- terminal 1) `node brokerRouterRouter 9000 9001`
- terminal 2) `node workerReq w1 localhost 9001`
- terminal 3) `node workerReq w2 localhost 9001`
- terminal 4) `node workerReq w3 localhost 9001`
- terminal 2) `ctrl-C`
- terminal 5) `node cliente A 3 localhost 9000 & node cliente B 3 localhost 9000 & node cliente C 3 localhost 9000`

Preguntas:

1. ¿Qué has observado en cuanto al funcionamiento de los clientes?

2.2.- El bróker tolerante a fallos (40 minutos)

A continuación presentamos el código fuente de un bróker que tolera los fallos de los workers. Observa detenidamente las estructuras de datos que se emplean y la gestión de los temporizadores para lograr que el bróker reenvíe peticiones a otro trabajador si alguno tarda demasiado en contestar.

broker.js

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
const ans_interval = 2000
lineaOrdenes("frontendPort backendPort")

let failed   = {}   // Map(worker:bool) failed workers has an entry
let working  = {}   // Map(worker:timeout) timeouts for workers executing tasks
let ready    = []   // List(worker) ready workers (for load-balance)
let pending  = []   // List([client,message]) requests waiting for workers
let frontend = zmq.socket('router')
let backend  = zmq.socket('router')

frontend.on('message', frontend_message)
backend.on('message', backend_message)
frontend.on('error'  , (msg) => {error(`${msg}`)})
backend.on('error'   , (msg) => {error(`${msg}`)})
process.on('SIGINT'  , adios([frontend, backend],"abortado con CTRL-C"))

creaPuntoConexion(frontend, frontendPort)
creaPuntoConexion(backend,  backendPort)

function dispatch(client, message) {
    traza('dispatch','client message',[client,message])
    if (ready.length) new_task(ready.shift(), client, message)
    else              pending.push([client,message])
}
function new_task(worker, client, message) {
    traza('new_task','client message',[client,message])
    working[worker] = setTimeout(failure(worker,client,message), ans_interval)
    backend.send([worker,'', client,message])
}
function failure(worker, client, message) {
    traza('failure','client message',[client,message])
    failed[worked] = true
    dispatch(client, message)
}
function frontend_message(client, sep, message) {
    traza('frontend_message','client sep message',[client,sep,message])
    dispatch(client, message)
}
function backend_message(worker, sep, client, message) {
    traza('backend_message','worker sep client message',[worker,sep,client,message])
    if (failed[worker]) return  // ignore messages from failed nodes
    if (worker in working) { // task response in-time
        clearTimeout(working[worker]) // cancel timeout
        delete(working[worker])
    }
    if (pending.length) new_task(worker, ...pending.shift())
    else ready.push(worker)
    if (client) frontend.send([client,'',message])
}
```

El código fuente lo tenemos en el directorio brokerToleranteFallos

- En dicho directorio encontrarás los ficheros cliente.js, servidorReq.js y broker.js
- cliente.js y servidorReq.js son idénticos a los utilizados en el patrón broker normal.

Preguntas:

1. Razona para qué se emplean cada una de las estructuras de datos: `failed, working, ready, pending`.
2. Trata de encontrar en el código la implementación del siguiente detalle relacionado con la tolerancia a fallos: “Se emplean temporizadores para sospechar que cierto worker ha fallado. Cuando tenemos esta firme sospecha, se reenvía la petición que estaba haciendo este worker a otro worker. Sin embargo, podría ocurrir que este worker que hemos supuesto que ha fallado, no había fallado, simplemente tardó demasiado tiempo en contestar”. Busca en el código fuente dónde y cómo se gestiona este tipo de “respuestas tardías”.

Prueba el bróker tolerante a fallos, realizando como mínimo la siguiente prueba:

Abrimos cinco terminales independientes y en todos ellos nos situamos en el directorio `brokerToleranteFallos:`

- terminal 1) `node broker.js 9000 9001`
- terminal 2) `node workerReq w1 localhost 9001`
- terminal 3) `node workerReq w2 localhost 9001`
- terminal 4) `node workerReq w3 localhost 9001`
- terminal 2) `ctrl-C`
- terminal 5) `node cliente A 3 localhost 9000 & node cliente B 3 localhost 9000 & node cliente C 3 localhost 9000`

Preguntas:

1. ¿Cuantas respuestas se obtienen?. Indica qué trabajadores las han enviado
2. ¿Quedan clientes esperando? Compara los resultados obtenidos respecto al brokerRouterRouter probado al comienzo de la sesión.
3. Este bróker únicamente aborda posibles fallos de workers. Indica qué posible estrategia podríamos implementar si quisiéramos también soportar fallos del bróker.