TSR: Examen de la Práctica 2

Nombre:		Apellidos:	
Grupo de prácticas:		Firma:	

Este examen consta de varias cuestiones de respuesta abierta. La puntuación asociada a cada cuestión se muestra en su enunciado respectivo.

ACTIVIDAD 1

En la primera sesión de la Práctica 2 se utilizó el patrón PUSH-PULL para desarrollar un sistema compuesto por tres tipos de componentes: origen, filtro y destino. Para el primer tipo, origen, se ofrecieron dos variantes: origen1 (que solo interactuaba con una instancia de filtro) y origen2 (que interactuaba con dos instancias de filtro). El tercer tipo, destino, únicamente mostraba en pantalla el contenido de los mensajes recibidos.

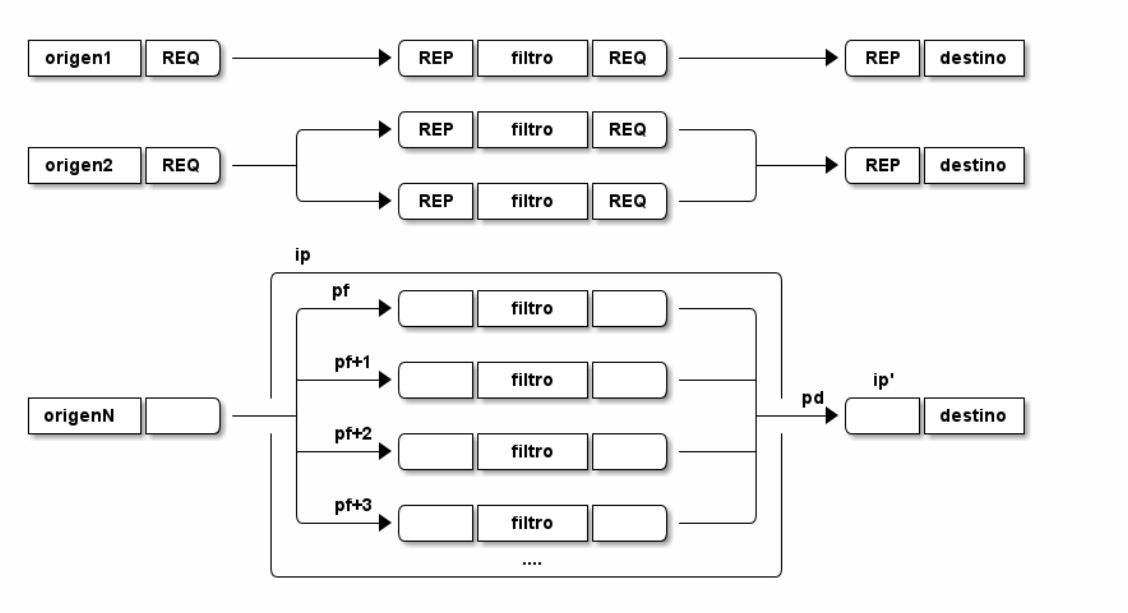
En esa misma sesión se ofrecían ejemplos de su utilización:

– terminal 1) **node origen1.js A localhost 9000** – terminal 2) **node filtro.js B 9000 localhost 8999 2** – terminal 3) **node destino.js C 8999**
– terminal 1) **node origen2.js A localhost 9000 localhost 9001** – terminal 2) **node filtro.js B 9000 localhost 8999 2** – terminal 3) **node filtro.js C 9001 localhost 8999 3** – terminal 4) **node destino.js D 8999**

El código del programa **filtro.js** se presenta seguidamente:

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion, conecta}= require('../tsr')
lineaOrdenes("nombre port hostSig portSig segundos")

let entrada = zmq.socket('pull')
let salida  = zmq.socket('push')

creaPuntoConexion(entrada, port)
conecta(salida, hostSig, portSig)

function procesaEntrada(emisor, iteracion) {
        traza('procesaEntrada','emisor iteracion',[emisor,iteracion])
        setTimeout(()=>{
                console.log(`Reenviado: [${nombre}, ${emisor}, ${iteracion}]`)
                salida.send([nombre, emisor, iteracion])
        }, parseInt(segundos)*1000)
}

entrada.on('message', procesaEntrada)
entrada.on('error' , (msg) => {error(`${msg}`)})
salida.on('error' , (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([entrada,salida],"abortado con CTRL-C"))
```

Responda las siguientes cuestiones, cuyos sistemas resultantes se muestran en esta figura:



1. (2 puntos) Revise con cuidado el código del programa **filtro.js** de esta actividad. Si se modificaran los programas **origen1.js**, **origen2.js** (ambos enviaban cuatro mensajes que finalmente recibía y mostraba el proceso destino) y **destino.js**, de manera que el socket utilizado en los programas de tipo **origen** fuera un REQ y el socket utilizado en el programa **destino.js** fuera un REP, explique si reemplazar el socket **entrada** de **filtro.js** por un REP y el socket **salida** de **filtro.js** por un REQ permitiría que el sistema resultante se comunicara sin problemas. Si fuera así, describa por qué. Si no fuera así, explique cuántos mensajes podrían llegar a **destino** y a qué se debería ese comportamiento.

La modificación descrita no permitirá que el sistema resultante se comunique como la hacía en su versión original. El patrón de comunicación PUSH-PULL es unidireccional y asincrónico. En él solo pueden enviarse mensajes utilizando el socket PUSH, mientras que el socket PULL únicamente podrá recibirlos. Por el contrario, el patrón REQ-REP es bidireccional sincrónico. Tanto REQ como REP pueden realizar ambas acciones: enviar y recibir. No obstante, REP necesita realizar en primer lugar una recepción, para enviar posteriormente una respuesta a la solicitud recibida.

Al sustituir el PUSH por REQ y el PULL por REP, el primer mensaje enviado por origen (tanto en origen1.js como en origen2.js) llegará a transmitirse y lo recibirá el REP del filtro correspondiente, que a su vez lo enviará y transmitirá con su REQ al componente destino, que también podrá recibirlo con su REP. Por tanto, el primer mensaje que envíe el componente origen llega al componente destino. Sin embargo, a partir de ese momento la comunicación queda bloqueada, ya que ninguno de los dos REQ utilizados podrá enviar otro mensaje y quedarán a la espera de alguna respuesta, pero esa respuesta no llegará jamás, pues el componente destino no respondía al filtro, ni el filtro respondía al componente origen.

Las cuatro operaciones send() del componente origen habrán retornado el control, pero los mensajes segundo, tercero y cuarto han quedado en la cola de envío de su socket REQ.

Para que llegara a transmitirse más de un mensaje desde el origen, deberíamos ampliar el código de los programas filtro.js y destino.js de manera que respondieran con algún mensaje cada vez que recibieran alguno. Eso no se describía en el enunciado de esta actividad y, en caso de aplicarse, estaría generando un modelo de comunicación bidireccional, que no respetaría el modelo de comunicaciones original.

2. (2 puntos) Desarrolle un programa **origenN.js** que utilizará un único socket para interactuar con N filtros, enviando 2N mensajes a ellos (sin pausas entre los envíos), que se estén ejecutando en una misma máquina, utilizando cada filtro un puerto distinto. Este programa debe recibir siempre estos argumentos: (1) el nombre que tendrá esa instancia del componente **origen**, (2) el número de filtros con los que interactuará, (3) la dirección IP (o nombre) del ordenador en el que residan los filtros y (4) el número del primer puerto utilizado por esos filtros, que emplearán números consecutivos.
Así, las siguientes líneas de órdenes generarían un sistema equivalente al mostrado previamente como ejemplo de uso del programa **origen2.js**:

```
– terminal 1) node origenN.js A 2 localhost 9000
– terminal 2) node filtro.js B 9000 localhost 8999 2
– terminal 3) node filtro.js C 9001 localhost 8999 3
– terminal 4) node destino.js D 8999
```

```
const {zmq, lineaOrdenes, traza, error, adios, conecta}= require('../tsr')
lineaOrdenes("nombre nFiltros host port")

let salida  = zmq.socket('push')
let numFiltros = parseInt(nFiltros) || 1

for (let i=0; i<numFiltros; i++)
        conecta(salida, host, parseInt(port)+i)

function enviarMensaje(emisor, iteracion){
        traza('enviarMensaje','emisor iteracion',[emisor,iteracion])
        salida.send([emisor, iteracion])
}

for (let i=1; i<=numFiltros*2; i++)
        enviarMensaje(nombre,i)

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

ACTIVIDAD 2

(4 puntos) En la última sesión de la Práctica 2, se presentó un broker tolerante a fallos que interactuaba con clientes y workers que utilizaban un socket REQ cada uno. Los programas **broker.js** y **cliente.js** correspondientes se muestran seguidamente:

```
 1: // broker.js
 2: const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} =
 3:     require('../tsr')
 4: const ans_interval = 2000 // deadline to detect worker failure
 5: lineaOrdenes("frontendPort backendPort")
 6: let failed   = {}     // Map(worker:bool) failed workers have an entry
 7: let working  = {}     // Map(worker:timeout) timeouts for workers executing tasks
 8: let ready    = []     // List(worker) ready workers (for load-balancing)
 9: let pending  = []     // List([client,message]) requests waiting for workers
10: let frontend = zmq.socket('router')
11: let backend  = zmq.socket('router')
12:
13: function dispatch(client, message) {
14:        traza('dispatch','client message',[client,message])
15:        if (ready.length) new_task(ready.shift(), client, message)
16:        else                    pending.push([client,message])
```

```
 17: }
 18: function new_task(worker, client, message) {
 19:        traza('new_task','client message',[client,message])
 20:        working[worker] = setTimeout(()=>{failure(worker,client,message)},
 21:            ans_interval)
 22:        backend.send([worker,'', client,'', message])
 23: }
 24: function failure(worker, client, message) {
 25:        traza('failure','client message',[client,message])
 26:        failed[worker] = true
 27:        dispatch(client, message)
 28: }
 29: function frontend_message(client, sep, message) {
 30:        traza('frontend_message','client sep message',[client,sep,message])
 31:        dispatch(client, message)
 32: }
 33: function backend_message(worker, sep1, client, sep2, message) {
 34:        traza('backend_message','worker sep1 client sep2 message',
 35:            [worker, sep1, client, sep2, message])
 36:        if (failed[worker]) return  // ignore messages from failed nodes
 37:        if (worker in working) { // task response in-time
 38:               clearTimeout(working[worker]) // cancel timeout
 39:               delete(working[worker])
 40:        }
 41:        if (pending.length) new_task(worker, ...pending.shift())
 42:        else ready.push(worker)
 43:        if (client) frontend.send([client,'',message])
 44: }
 45:
 46: frontend.on('message', frontend_message)
 47:  backend.on('message', backend_message)
 48: frontend.on('error'  , (msg) => {error(`${msg}`)})
 49:  backend.on('error'  , (msg) => {error(`${msg}`)})
 50:  process.on('SIGINT' , adios([frontend, backend],"abortado con CTRL-C"))
 51:
 52: creaPuntoConexion(frontend, frontendPort)
 53: creaPuntoConexion( backend,  backendPort)
```

```
 1: // cliente.js
 2: const {zmq, lineaOrdenes, traza, error, adios, conecta} =
 3:     require('../tsr')
 4: lineaOrdenes("id brokerHost brokerPort")
 5: let req = zmq.socket('req')
 6: req.identity = id
 7: conecta(req, brokerHost, brokerPort)
 8: req.send(id)
 9:
10: function procesaRespuesta(msg) {
11:        traza('procesaRespuesta','msg',[msg])
12:        adios([req], `Recibido: ${msg}. Adios`)()
13: }
14: req.on('message', procesaRespuesta)
15: req.on('error', (msg) => {error(`${msg}`)})
16: process.on('SIGINT', adios([req],"abortado con CTRL-C"))
```

Se solicita transformar estos programas **broker.js** y **cliente.js** para que el fallo de los workers deje de ser transparente. Para ello, cuando el cliente reciba su respuesta recibirá un primer segmento adicional de tipo Booleano. Si este primer segmento tiene un valor **true**, indica que la solicitud ha podido procesarse y ha obtenido una respuesta, contenida en el siguiente segmento. Entonces, el cliente finalizará tras mostrar en pantalla una línea "**Recibido: XYZ. Adios**", donde el texto **XYZ** deberá ser realmente el contenido del mensaje de respuesta. Observe que para gestionar este nuevo segmento, el broker necesitará extender **en todos los casos** el contenido de los mensajes que envíe desde su socket frontend. Si por el contrario se recibiese un valor

false en el primer segmento, eso indicaría que el trabajador que fue elegido para procesar la petición ha fallado. Entonces el cliente reenviará su petición de inmediato y esperará respuesta.

Describa y programe las modificaciones que debería aplicar en ambos componentes. No es necesario que los reescriba por completo. Basta con indicar qué variables globales o funciones necesitarían cambios, escribiendo únicamente el código correspondiente a esos elementos.

En el programa cliente.js, se debe modificar la función procesaRespuesta() para que quede de la siguiente manera:

```
function procesaRespuesta(ok, msg) {
    traza('procesaRespuesta','msg',[msg])
    if (ok+"" == "true")
        adios([req], `Recibido: ${msg}. Adios`)()
    else req.send(id)
}
```

Obsérvese que se ha necesitado un nuevo primer parámetro en la función procesaRespuesta(), de manera que en él se reciba el valor booleano que indica si ha podido procesarse la solicitud sin errores. Posteriormente, se necesita una línea para comprobar el valor recibido en ese primer segmento de la respuesta. Si ese valor es **true**, mantenemos el comportamiento anterior. En otro caso, se envía de nuevo la misma petición al broker. Por ello, algo después debería obtenerse otra respuesta, que gestionaremos de la misma manera.

Por su parte, en el programa broker.js hay que realizar estas modificaciones:

- La línea 43 original contenía la instrucción de envío de la respuesta al cliente. Ese mensaje debe seguir enviándose, pero ahora incluirá un segmento adicional, justo antes del último, con la constante **true**. Por tanto, quedará así:

  ```
  if (client) frontend.send([client, '', true, message])
  ```

- Cuando vencía el *timeout* establecido para la gestión de una petición, el worker correspondiente era marcado como caído y esa petición era gestionada de nuevo con dispatch(). Todo eso se hacía en la función failure(), pero ahora esa función debe cambiar para no utilizar dispatch() sino devolver una respuesta negativa al cliente, con su primer segmento entregado al cliente a **false** (será el tercero en esta instrucción de envío, pues el primero lo necesita el socket emisor de tipo ROUTER para identificar la conexión a utilizar en esa operación send() y el segundo debe ser un delimitador, puesto que el cliente está utilizando un socket REQ y esos sockets necesitan que los mensajes recibidos tengan una cadena vacía en su primer segmento). Por tanto, bastará con sustituir la línea 27 original y dejarla como sigue (el cuarto segmento puede no utilizarse, o rellenarse con cualquier contenido, pues no debería ser tratado por el cliente):

  ```
  frontend.send([cliente,'',false])
  ```

ACTIVIDAD 3

(2 puntos) En la tercera sesión de la Práctica 2 se solicitó la división del broker ROUTER-ROUTER en dos mitades: broker1 (que gestionaría las peticiones de los clientes) y broker2 (que gestionaría los workers disponibles). Esas dos mitades necesitan comunicarse. Indique qué sockets ha utilizado para realizar esa comunicación. Justifique por qué considera que esa combinación es la más razonable. Para ello, demuestre que la comunicación será posible en todos los casos, no se perderán peticiones de los clientes y cada cliente recibirá la respuesta que le correspondía.

Esta cuestión admite múltiples soluciones, pues no hay grandes diferencias entre ellas por lo que respecta a eficiencia o robustez.

Se podría haber empleado un socket DEALER en cada mitad, pues este tipo de socket es bidireccional y asincrónico, sin restricciones en cuanto a iniciar el envío de información. El socket DEALER plantearía problemas en caso de que hubiera múltiples instancias de cualquiera de las dos mitades, pues entonces seguiría una estrategia circular en el envío de mensajes hacia esas múltiples instancias y esto conduciría a perder algún mensaje, puesto que quizá se enviaran a una instancia que no podría gestionarlos. Sin embargo, en la tercera sesión de esta práctica 2 no se planteaba (al menos, no inicialmente) que pudiera haber más de una instancia de cada mitad.

Si solo hay un broker1 y otro broker2, y ambos emplean un socket DEALER para intercomunicarse, broker1 y broker2 guardarán las peticiones pendientes y los workers disponibles, respectivamente. Para ello, alguno de los dos mantendrá un contador del número de "elementos" (léase, peticiones o workers) que guarda el otro. La gestión de ese contador deberá basarse en mensajes transmitidos desde la otra mitad. Como los sockets DEALER son asincrónicos y bidireccionales, ese intercambio de mensajes estará permitido en todos los casos, por lo que no habrá problema. Si el contador correspondiente está bien gestionado, no habrá riesgo de perder peticiones pendientes (que pudieran llegar a broker2 antes de que este mantuviera algún worker preparado para procesar peticiones), por lo que no se perderán mensajes con este diseño. Por otra parte, la gestión de las respuestas, para entregarlas al cliente correcto estará basada en la transferencia y mantenimiento del segmento que identifica al cliente, añadido automáticamente en la recepción realizada en el socket frontend de broker1. Ese segmento se podrá enviar con el resto de segmentos de la petición, y recibirse después en la correspondiente respuesta, pues los sockets DEALER pueden gestionar fácilmente mensajes multisegmento.

Otra solución equivalente reemplazaría este único canal DEALER-DEALER por dos canales unidireccionales PUSH-PULL. Uno permitiría el envío desde broker1 a broker2, mientras otro lo haría en sentido contrario. Esta combinación sería funcionalmente equivalente a un único canal DEALER-DEALER, por lo que la descripción realizada previamente también sería aplicable aquí.

Una tercera solución que también llegaría a funcionar utilizaría un socket ROUTER en broker1 y un socket DEALER en broker2. Sin embargo, en este caso la comunicación siempre debería iniciarla broker2, para que de esta manera el ROUTER de broker1 sepa qué identidad utilizar en el primer segmento de los mensajes que envíe hacia broker2. Esto no sería una restricción grave si cada mensaje de registro inicial enviado por un trabajador fuera retransmitido por broker2 hacia broker1, de manera que broker1 incrementase el contador de workers disponibles.

También podría haberse empleado la estrategia inversa: ROUTER en broker2 y DEALER en broker1. En este caso, sería broker2 quien mantendría un contador para saber cuántas peticiones pendientes existen en cada momento. No obstante, esta solución podría necesitar más mensajes para gestionar cada interacción entre clientes y workers.

Como puede observarse, existen múltiples alternativas. En todas ellas se retransmite la identidad del cliente o del worker, obtenidas en la recepción del mensaje correspondiente en los sockets frontend y backend de broker1 y broker2 para mantener correctamente qué peticiones están pendientes y qué cliente originó cada una de ellas, o bien qué workers están disponibles cuando no haya peticiones pendientes.

Las soluciones basadas en sockets REQ y REP, aunque pudieran funcionar, no serían recomendables, pues bloquearían innecesariamente la transferencia de información entre broker1 y broker2. Por ejemplo, se complicaría la transferencia de múltiples peticiones a múltiples workers si las operaciones a ejecutar fueran prolongadas: la segunda petición no podría transmitirse a broker2 hasta que la primera respuesta hubiera llegado a broker1.