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.

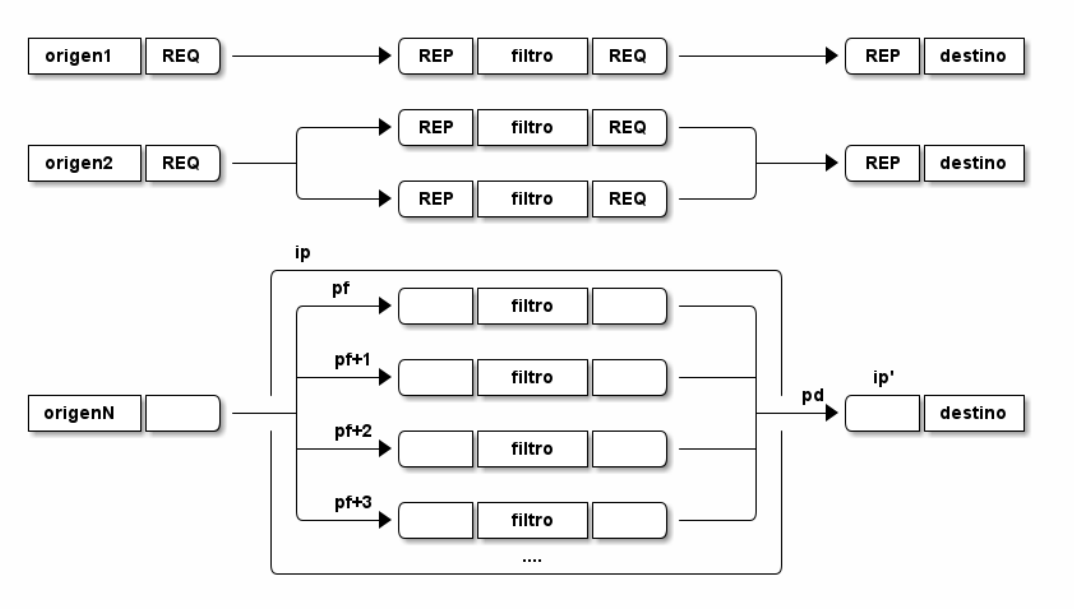
En esa misma sesión se ofrecían ejemplos de su utilización:

– 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.

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**

ACTIVIDAD 2

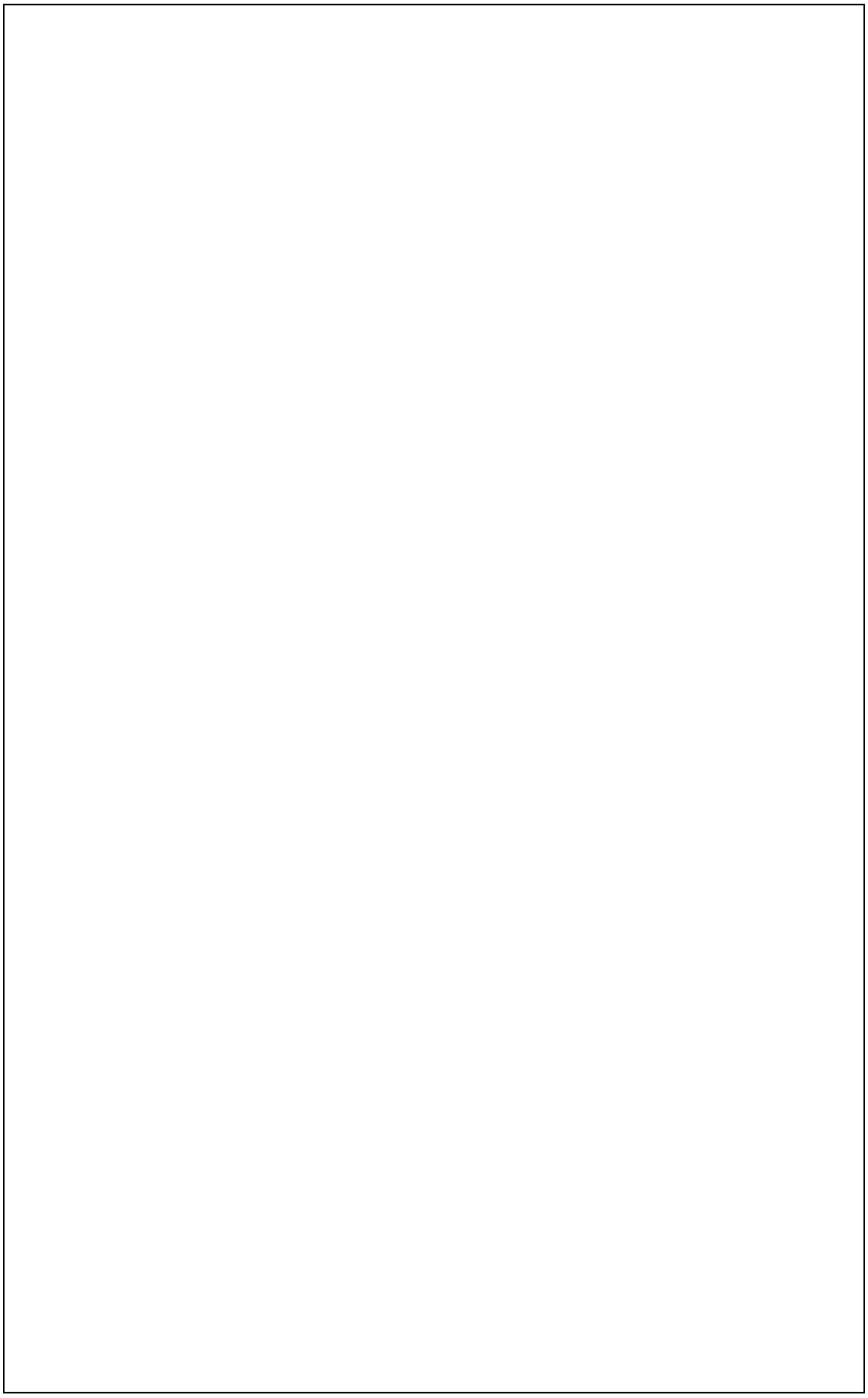
(4 puntos) En la última sesión de la Práctica 2, se presentó un broker tolerante a fallos que interactuaba con clientes y workers que utilizaban un socket REQ cada uno. Los programas **broker.js** y **cliente.js** correspondientes se muestran seguidamente:

```
 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.



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.