TSR: Examen de la Pràctica 2

Nom:		Cognoms:	
Grup de pràctiques:		Signatura:	

Aquest examen consta de diverses qüestions de resposta oberta. La puntuació associada a cada qüestió es mostra en el seu enunciat respectiu.

ACTIVITAT 1

A la primera sessió de la Pràctica 2 es va utilitzar el patró PUSH-PULL per desenvolupar un sistema compost per tres tipus de components: **origen**, **filtro** (filtre) i **destino** (destinació). Per al primer tipus, origen, es van oferir dues variants: **origen1** (que només interactuava amb una instància de filtre) i **origen2** (que interactuava amb dues instàncies de filtre). El tercer tipus, **destino**, només mostrava a la pantalla el contingut dels missatges rebuts.

En aquesta sessió s'oferien exemples d'utilització:

– 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 codi del programa **filtro.js** es mostra a continuació:

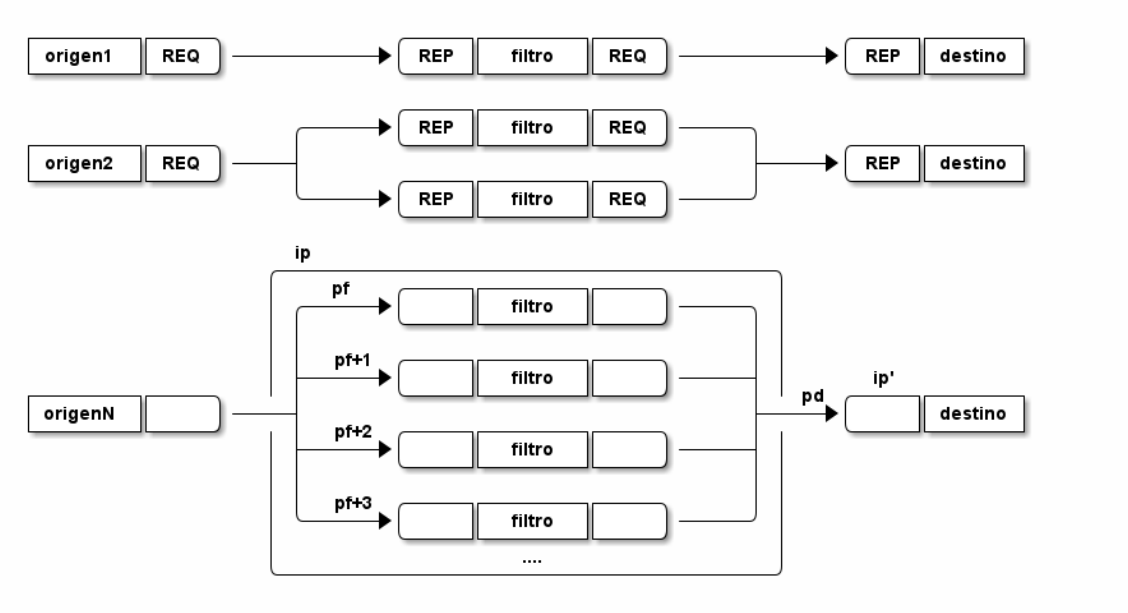
```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion, conecta} = require('../tsr')
lineaOrdenes("nombre port hostSig portSig segundos")

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"))
```

Responeu les qüestions següents, els sistemes resultants de les quals es mostren en aquesta figura:



1. (2 punts) Reviseu amb cura el codi del programa **filtro.js** d'aquesta activitat. Si es modificaren els programes **origen1.js**, **origen2.js** (tots dos enviaven quatre missatges que finalment rebia i mostrava el procés destinació) i **destino.js**, de manera que el socket utilitzat als programes de tipus **origen** fos un REQ i el socket utilitzat al programa **destino.js** fos un REP, expliqueu si reemplaçar el socket **entrada** de **filtro.js** per un REP i el socket **salida** de **filtro.js** per un REQ permetria que el sistema resultant es comunicara sense problemes. Si fóra així, descriviu per què. Si no fóra així, expliqueu quants missatges podrien arribar a **destino** i a què es deuria aquest comportament.

2. (2 punts) Desenvolupeu un programa **origenN.js** que utilitzarà un únic socket per interactuar amb N filtres, enviant 2N missatges a ells (sense pauses entre els enviaments), que s'estiguen executant en una mateixa màquina, utilitzant cada filtre un port diferent. Aquest programa ha de rebre sempre aquests arguments: (1) el nom que tindrà aquesta instància del component **origen**, (2) el nombre de filtres amb els que interactuarà, (3) l'adreça IP (o nom) de l'ordinador on estan els filtres i (4) el número del primer port utilitzat per aquests filtres, que empraran números consecutius.
   Així, les línies d'ordres següents generarien un sistema equivalent al mostrat prèviament com a exemple d'ús 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**

ACTIVITAT 2

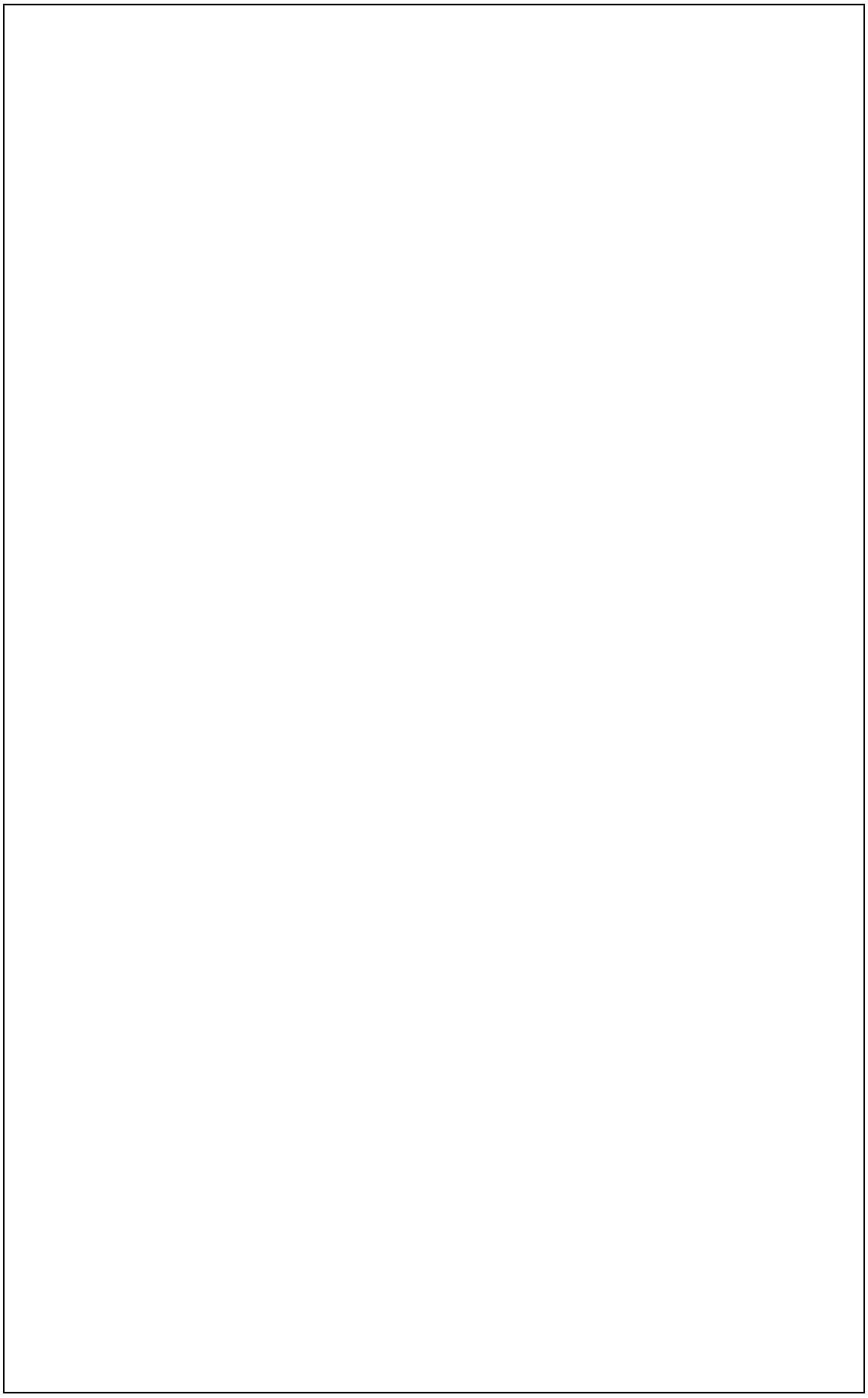
(4 punts) A la darrera sessió de la Pràctica 2, es va presentar un broker tolerant a fallades que interactuava amb clients i workers que utilitzaven un socket REQ cadascun. Els programes **broker.js** i **cliente.js** corresponents són:

```
 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 sol·licita transformar aquests programes **broker.js** i **cliente.js** perquè la decisió dels workers deixe de ser transparent. Per a fer això, quan el client reba la seua resposta rebrà un primer segment addicional de tipus Booleà. Si aquest primer segment té un valor **true**, indica que la sol·licitud s'ha pogut processar i ha obtingut una resposta, continguda al segment següent. Aleshores, el client finalitzarà després de mostrar en pantalla una línia "**Recibido: XYZ. Adios**", on el text **XYZ** haurà de ser realment el contingut del missatge de resposta. Observeu que per gestionar aquest nou segment, el broker necessitarà estendre **en tots els casos** el contingut dels missatges que s'envien des del socket frontend. Si per contra es rebera un valor **false** al primer segment, això indicaria que el treballador que va ser elegit per processar la petició ha fallat. Aleshores el client reenviarà la seva petició immediatament i esperarà resposta.

Descriviu i programeu les modificacions que cal aplicar en ambdós components. No cal que els reescriviu del tot. Només cal indicar quines variables globals o funcions necessitarien canvis, escrivint únicament el codi corresponent a aquests elements.



ACTIVITAT 3

(2 punts) A la tercera sessió de la Pràctica 2 es va sol·licitar la divisió del broker ROUTER-ROUTER en dues meitats: broker1 (que gestionaria les peticions dels clients) i broker2 (que gestionaria els workers disponibles). Aquestes dues meitats necessiten comunicar-se. Indiqueu quins sockets heu utilitzat per realitzar aquesta comunicació. Justifiqueu per què considereu que aquesta combinació és la més raonable. Per això, demostreu que la comunicació serà possible en tots els casos, no es perdran peticions dels clients i cada client rebrà la resposta que li corresponia.