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**

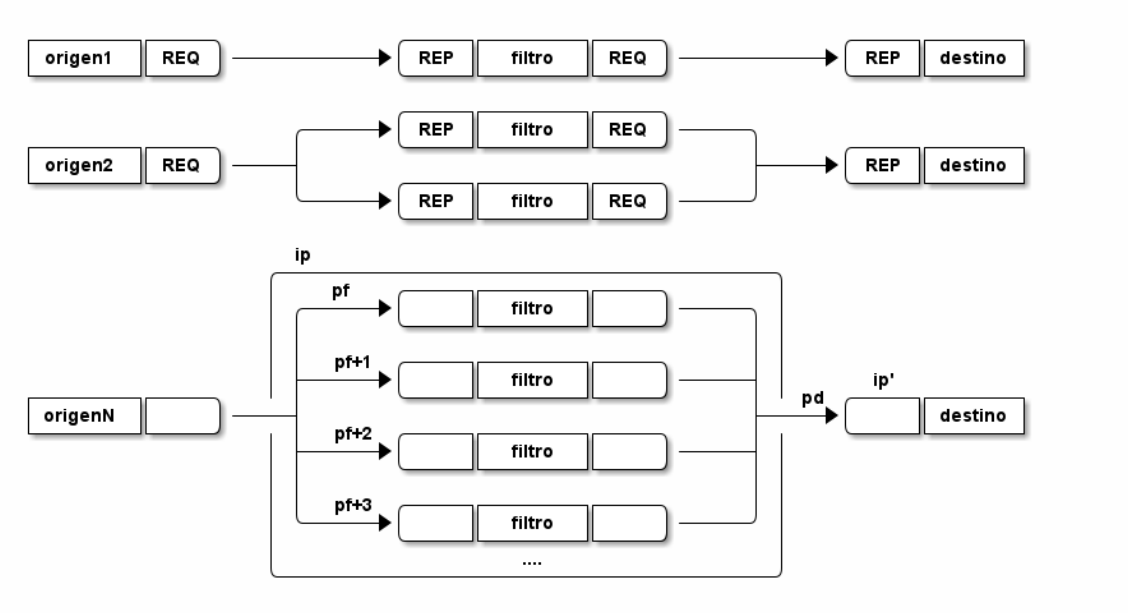
El codi del programa **filtro.js** es mostra a continuació:

```
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.

La modificació descrita no permetrà que el sistema resultant es comunique com ho feia en la versió original. El patró de comunicació PUSH-PULL és unidireccional i asincrònic. Només es poden enviar missatges utilitzant el socket PUSH, mentre que el socket PULL únicament els podrà rebre. Per contra, el patró REQ-REP és bidireccional sincrònic. Tant REQ com REP poden fer ambdues accions: enviar i rebre. Tot i això, REP necessita realitzar en primer lloc una recepció, per enviar posteriorment una resposta a la sol·licitud rebuda.

En substituir el PUSH per REQ i el PULL per REP, el primer missatge enviat per origen (tant a origen1.js com a origen2.js) arribarà a transmetre's i el rebrà el REP del filtre corresponent, que al seu torn l'enviarà i transmetrà amb el REQ al component destí, que també podrà rebre'l amb el REP. Per tant, el primer missatge que envie el component origen arriba al component destí. Tot i això, a partir d'aquest moment la comunicació queda bloquejada, ja que cap dels dos REQ utilitzats podrà enviar un altre missatge i quedaran a l'espera d'alguna resposta, però aquesta resposta no arribarà mai, ja que el component destí no responia al filtre, ni el filtre responia al component origen.

Els quatre send() del component origen hauran retornat el control, però els missatges segon, tercer i quart han quedat a la cua d'enviament del seu socket REQ.

Per tal que s'arribara a transmetre més d'un missatge des de l'origen, hauríem d'ampliar el codi dels programes filtro.js i destino.js de manera que respongueren amb algun missatge cada vegada que en reberen algun. Això no es descrivia a l'enunciat d'aquesta activitat i, en cas d'aplicar-se, estaria generant un model de comunicació bidireccional, que no respectaria el model de comunicacions original.

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**

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

let eixida  = zmq.socket('push')
let numFiltres = parseInt(nFiltres) || 1

for (let i=0; i<numFiltres; i++)
        conecta(eixida, host, parseInt(port)+i)

function enviarMissatge(emissor, iteracio) {
        traza('enviarMissatge','emissor iteracio',[emissor,iteracio])
        eixida.send([emissor, iteracio])
}

for (let i=1; i<=numFiltres*2; i++)
        enviarMissatge(nom,i)

 eixida.on('error' , (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([eixida],"avortat amb CTRL-C"))
```

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.

Al programa cliente.js, s'ha de modificar la funció procesaRespuesta() perquè quede de la següent manera:

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

Observeu que s'ha necessitat un nou primer paràmetre en la funció procesaRespuesta(), de manera que es reba el valor booleà que indica si s'ha pogut processar la sol·licitud sense errors. Posteriorment, cal una línia per comprovar el valor rebut en aquest primer segment de la resposta. Si aquest valor és **true**, mantenim el comportament anterior. En qualsevol altre cas, s'envia de nou la mateixa petició al broker. Per això, poc després caldria obtenir una altra resposta, que gestionarem de la mateixa manera.

Per altra banda, al programa broker.js cal fer aquestes modificacions:

- La línia 43 original contenia la instrucció per a respondre al client. Aquest missatge ha de continuar enviant-se, però ara inclourà un segment addicional, just abans del darrer, amb la constant **true**. Per tant, queda així:

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

- Quan vencia el *timeout* establert per a la gestió d'una petició, el worker corresponent era marcat com a caigut i aquesta petició era gestionada de nou amb dispatch(). Tot això es feia a la funció failure(), però ara aquesta funció ha de canviar per no utilitzar dispatch() sinó tornar una resposta negativa al client, amb el seu primer segment entregat al client a **false** (serà el tercer en aquesta instrucció d'enviament, ja que el primer el necessita el socket emissor de tipus ROUTER per identificar la connexió a utilitzar en aquesta operació send() i el segon ha de ser un delimitador, ja que el client està utilitzant un socket REQ i aquests sockets necessiten que els missatges rebuts tinguen una cadena buida en el primer segment). Per tant, només cal substituir la línia 27 original i deixar-la com segueix (el quart segment pot no utilitzar-se, o emplenar-se amb qualsevol contingut, ja que no hauria de ser tractat pel client):

  frontend.send ([client,'', **false**])

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.

Aquesta qüestió admet múltiples solucions, ja que no hi ha grans diferències entre elles pel que fa a eficiència o robustesa.

Es podria haver emprat un socket DEALER a cada meitat, ja que aquest tipus de socket és bidireccional i asincrònic, sense restriccions quant a iniciar l'enviament d'informació. El socket DEALER plantejaria problemes en cas que hi haguera múltiples instàncies de qualsevol de les dues meitats, doncs llavors seguiria una estratègia circular en l'enviament de missatges cap a aquestes múltiples instàncies i això conduiria a perdre algun missatge, ja que potser s'enviaren a una instància que no els podria gestionar. Tot i això, a la tercera sessió d'aquesta pràctica 2 no es plantejava (almenys, no inicialment) que hi poguera haver més d'una instància de cada meitat.

Si només hi ha un broker1 i un altre broker2, i tots dos fan servir un socket DEALER per intercomunicar-se, broker1 i broker2 guardaran les peticions pendents i els workers disponibles, respectivament. Per això, algun dels dos mantindrà un comptador del nombre d'elements (és a dir, peticions o workers) que guarda l'altre. La gestió del comptador s'ha de basar en missatges transmesos des de l'altra meitat. Com que els sockets DEALER són asincrònics i bidireccionals, aquest intercanvi de missatges estarà permès en tots els casos, per la qual cosa no hi haurà cap problema. Si el comptador corresponent està ben gestionat, no hi haurà risc de perdre peticions pendents (que arribaren a broker2 abans que aquest mantinguera algun worker preparat per processar peticions), per la qual cosa no es perdran missatges amb aquest disseny. D'altra banda, la gestió de les respostes, per lliurar-les al client correcte estarà basada en la transferència i manteniment del segment que identifica el client, afegit automàticament a la recepció realitzada al socket frontend de broker1. Aquest segment es podrà enviar amb la resta de segments de la petició, i rebre's després en la resposta corresponent, ja que els sockets DEALER poden gestionar fàcilment missatges multisegment.

Una altra solució equivalent reemplaçaria aquest únic canal DEALER-DEALER per dos canals unidireccionals PUSH-PULL. Un permetria l'enviament des de broker1 a broker2, mentre que un altre ho faria en sentit contrari. Aquesta combinació seria funcionalment equivalent a un

únic canal DEALER-DEALER, per la qual cosa la descripció feta prèviament també seria aplicable ara.
Una tercera solució que també arribaria a funcionar utilitzaria un socket ROUTER a broker1 i un socket DEALER a broker2. No obstant això, en aquest cas la comunicació sempre hauria d'iniciar-la broker2, perquè així el ROUTER de broker1 sàpiga quina identitat utilitzar en el primer segment dels missatges que envie cap a broker2. Això no seria una restricció greu si cada missatge de registre inicial enviat per un treballador fóra retransmès per broker2 cap a broker1, de manera que broker1 incrementara el comptador de workers disponibles.
També podria haver-se emprat l'estratègia inversa: ROUTER a broker2 i DEALER a broker1. En aquest cas, seria broker2 qui mantindria un comptador per saber quantes peticions pendents hi ha a cada moment. Això no obstant, aquesta solució podria necessitar més missatges per gestionar cada interacció entre clients i workers .
Com es pot observar, hi ha múltiples alternatives. En totes es retransmet la identitat del client o del worker, obtingudes a la recepció del missatge corresponent als sockets frontend i backend de broker1 i broker2 per mantenir correctament quines peticions estan pendents **i quin client va originar cadascuna** (i això ens assegura que les respostes aniran dirigides al client que va enviar la petició associada), o bé quins workers estan disponibles quan no hi haja sol·licituds pendents.
Les solucions basades en sockets REQ i REP, encara que puguen funcionar, no serien recomanables, ja que bloquejarien innecessàriament la transferència d'informació entre broker1 i broker2. Per exemple, es complicaria la transferència de múltiples peticions a múltiples workers si les operacions a executar fossen prolongades: la segona petició no es podria transmetre a broker2 fins que la primera resposta haguera arribat a broker1.