NIST: Lab 2

Name:		Surname:	
Lab group:		Signature:	

This exam consists of several open-ended questions. The grade associated with each question is shown in its respective statement.

ACTIVITY 1

In the first session of Lab 2, the PUSH-PULL pattern was used to develop a system composed of three types of components: **origen** (source), **filtro** (filter), and **destino** (destination). For the first type, **origen**, two variants were provided: **origen1** (which only interacted with one filter instance) and **origen2** (which interacted with two filter instances). The third type, **destino**, only displayed the content of received messages on the screen.

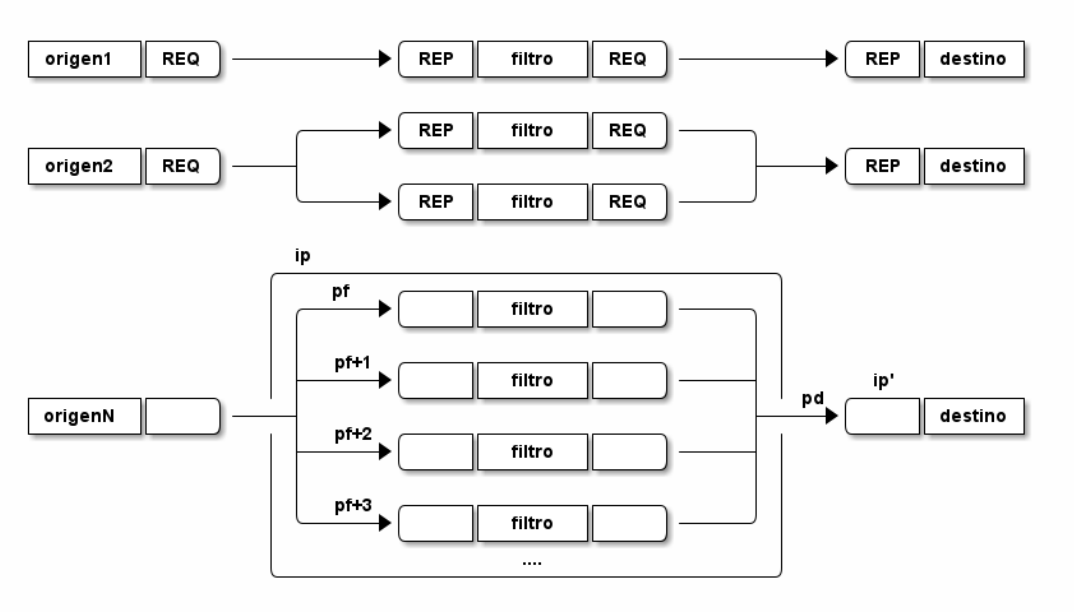
Several examples of their use are:

– 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**

The code of **filtro.js** is:

```
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"))
```

Answer the following questions, whose resulting systems are shown in this figure:



1. (2 points) Carefully review the program **filtro.js**. If the programs **origen1.js**, **origen2.js** (both send four messages, to be received and displayed by the destination process) and **destino.js** were modified, so that the socket used in the programs of type **origen** was a REQ and the socket used in **destino.js** was a REP, explain whether replacing the socket **entrada** of **filtro.js** with a REP and the socket **salida** of **filtro.js** with a REQ would allow the resulting system to communicate without problems. If so, describe why. If not, explain how many messages reach **destino** and why this behaviour arises.

2. (2 points) Develop a program **origenN.js** that uses a single socket to interact with N filters, and sends 2N messages to them (with no pauses between sends). Those filters (i.e., processes that run **filtro.js**) run on the same computer, and each filter uses a different port. The new program must always receive these arguments: (1) name, (2) the number of filters to interact with, (3) the IP address (or name) of the computer where filters run, and (4) the number of the first port used by those filters, which will use consecutive numbers.

   Thus, the following command lines would generate a system equivalent to the one previously shown as an example of use of **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**

ACTIVITY 2

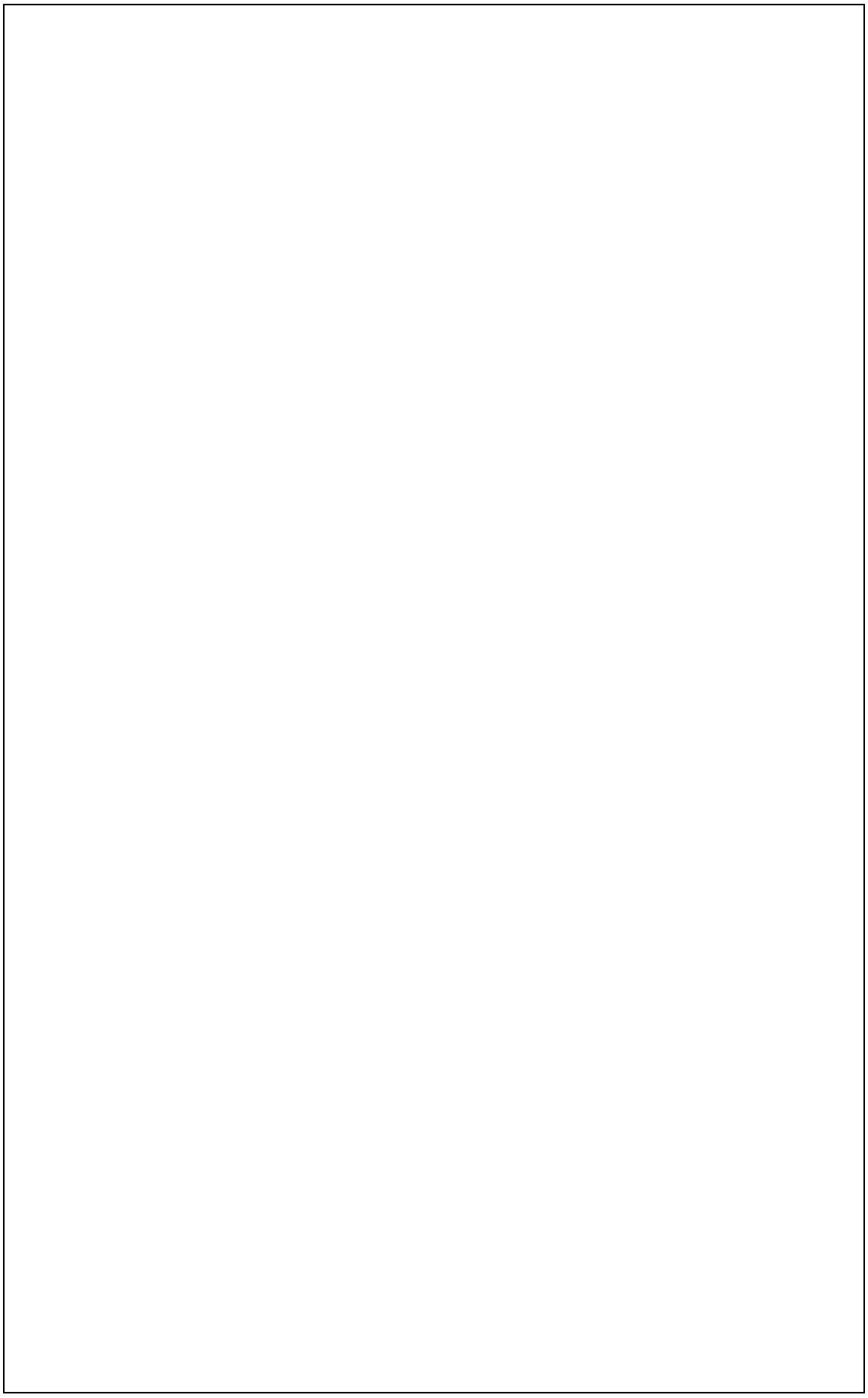
(4 points) In the last session of Lab 2, a fault-tolerant broker was presented. It interacts with clients and workers that use one REQ socket each one. The corresponding **broker.js** and **client.js** programs are shown below:

```
 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"))
```

Please, transform these **broker.js** and **cliente.js programs** so that the failure of the workers is no longer transparent. To this end, when the client receives its response, it will receive a first additional segment of Boolean type. If this first segment is **true**, it indicates that the request was able to be processed and got a response, contained in the next segment. If so, the client finishes after displaying on the screen a line "**Recibido: XYZ. Adios**", where the text **XYZ** should actually be the content of the reply message. Note that to handle this new segment, the broker will need to extend the content of the messages it sends from its frontend socket **in all cases**. If, on the contrary, the first answer segment is **false**, then the worker that was chosen to process the request has failed. If so, the client forwards again its request immediately and waits for a response.

Describe and write the modifications that you should apply to both components. You don't need to rewrite those programs completely. Instead, indicate which global variables or functions need changes, writing only the code corresponding to those elements.



ACTIVITY 3

(2 points) The third session of Lab 2 requests the division of the ROUTER-ROUTER broker into two halves: broker1 (which manages client requests) and broker2 (which manages available workers). Those two halves need to communicate. Indicate which sockets you have used to perform that communication. Justify why you consider that this combination is the most reasonable. To do this, demonstrate that communication is possible in all cases, client requests are not lost, and each client receives the response that corresponds to it.