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.

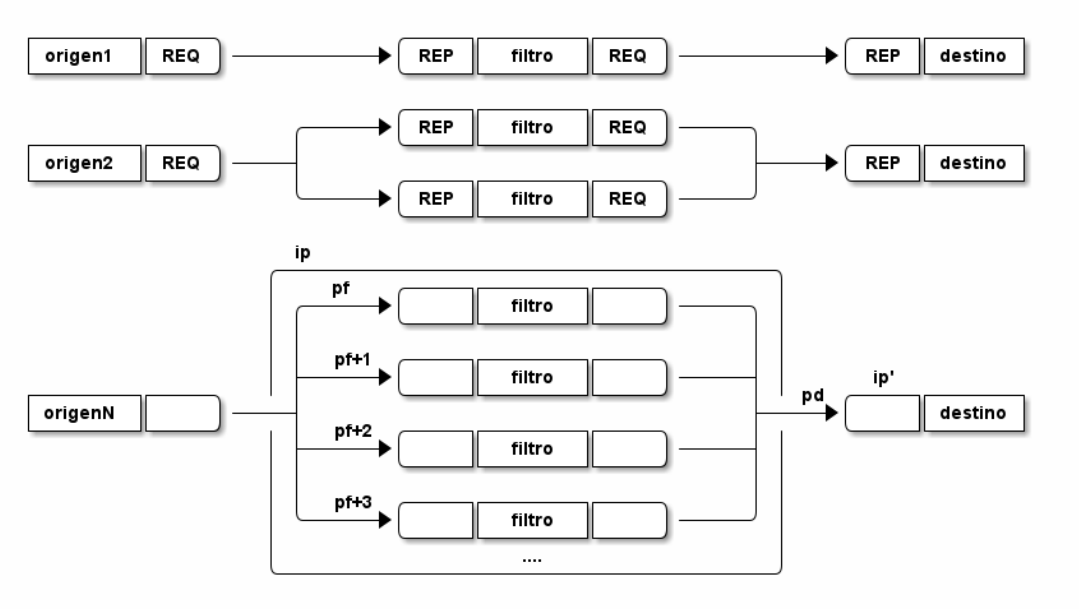
Several examples of their use are:

– 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.

   The modification described will not allow the resulting system to communicate as it did in its original version. The PUSH-PULL communication pattern is unidirectional and asynchronous. In it, PUSH sockets only send messages, while the PULL socket can only receive them. By contrast, the REQ-REP pattern is synchronous bidirectional. Both REQ and REP can perform both actions: send and receive. However, REP needs to perform a reception first, in order to subsequently send a response to the received request.
   By replacing the PUSH with REQ and the PULL with REP, the first message sent by **origen** (both in origen1.js and in origen2.js) will be transmitted and will be received by the REP of the corresponding filter, which in turn will send and transmit it with its REQ to the destination component, which can also receive it with its REP. Therefore, the first message sent by **origen** reaches **destino**. However, from that moment on the communication is blocked, since neither of the two REQs used will be able to send another message and they will be waiting for a response, but that response will never arrive, since **destino** does not respond to **filtro**, nor **filtro** to **origen**.
   The four send() operations of **origen** return control, but the second, third, and fourth messages have been left in the sending queue of its REQ socket.
   In order for more than one message to be transmitted from **origen**, we would need to extend the code of filtro.js and destino.js so that they respond with some message each time they receive one. This was not described in the statement of this activity and, if applied, it would be generating a bidirectional communication model, which would not respect the original communications model.

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
```

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

let output  = zmq.socket('push')
let numFilters = parseInt(nFilters) || 1

for (let i=0; i<numFilters; i++)
        conecta(output, host, parseInt(port)+i)

function sendMessage(sender, iteration) {
        traza('sendMessage','sender iteration',[sender,iteration])
        output.send([sender, iteration])
}

for (let i=1; i<=numFilters*2; i++)
        sendMessage(name,i)

 output.on('error'  , (msg) => {error(`${msg}`)})
process.on('SIGINT' , adios([output],"aborted with CTRL-C"))
```

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.

In the cliente.js program, you must modify the function procesaRespuesta() so that it reads as follows:

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

Note that a new first parameter has been needed in the function procesaRespuesta(), so that the boolean value that indicates whether the request could be processed without errors is received there. Subsequently, a line is needed to check the value received in that first segment of the response. If that value is **true**, we keep the old behaviour. Otherwise, the same request is sent to the broker again. Therefore, a little later another response should be obtained, which we will manage in the same way.

For its part, in the broker.js program you have to make these modifications:

- The original line 43 contained the instruction to send the response to the client. That message should still be sent, but it will now include an extra segment, just before the last one, with the constant **true**. Therefore, it will be like this:

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

- When the *timeout* established for the management of a request expired, the corresponding worker was marked as down and that request was managed again with dispatch(). All of that was done in the failure() function, but now that function must change to not use dispatch() but instead return a negative response to the client, with its first segment delivered to the client set to **false** (it will be the third in this statement, because the first one is needed by the sending socket of the ROUTER type to identify the connection to be used in that send() operation and the second must be a delimiter, since the client is using a REQ socket and those sockets need that received messages have an empty string in its first segment). Therefore, it will suffice to replace the original line 27 and leave it as follows (the fourth segment can be left unused, or filled with any content, as it should not be processed by the client):

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

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.

This issue admits multiple solutions, since there are no major differences between them in terms of efficiency or robustness. We discuss some of them.

A DEALER socket could have been used in each half, since this type of socket is bidirectional and asynchronous, with no restrictions on starting to send information. The DEALER socket would pose problems if there were multiple instances of either halves, as it would then follow a circular strategy in sending messages to those multiple instances and this would lead to missing messages, since they might be sent to one instance that couldn't manage them. However, in the third session of this practice 2 it was not considered (at least, not initially) that there could be more than one instance of each half.

If there is only one broker1 and one broker2, and they both use a DEALER socket to communicate, broker1 and broker2 will hold pending requests and available workers, respectively. To do this, one of the two will keep a counter of the number of “elements” (i.e. requests or workers) that the other keeps. The management of that counter should be based on messages transmitted from the other half. Since DEALER sockets are asynchronous and bidirectional, this exchange of messages will be allowed in all cases, so there will be no problem. If the corresponding counter is well managed, there will be no risk of losing pending requests (which could reach broker2 before it maintains any worker ready to process requests), so no messages will be lost with this design. On the other hand, the management of the responses, in order to deliver them to the correct client, will be based on the transfer and maintenance of the segment that identifies the client, automatically added in the reception made in the frontend socket of broker1. This segment can be sent with the rest of the request segments, and later received in the corresponding response, since DEALER sockets can easily handle multi-segment messages.

Another equivalent solution would replace this single DEALER-DEALER channel with two unidirectional PUSH-PULL channels. One would allow sending from broker1 to broker2, while another would do the opposite. This combination would be functionally equivalent to a single DEALER-DEALER channel, so the description previously made would also apply here.

A third solution that would also work would use a ROUTER socket on broker1 and a DEALER socket on broker2. However, in this case the communication should always be initiated by

broker2, so that broker1's ROUTER knows which identity to use in the first segment of messages it sends to broker2. This would not be a hard restriction if every initial registration message sent by a worker was relayed by broker2 to broker1, so that broker1 incremented the available workers counter.

The reverse strategy could also have been used: ROUTER on broker2 and DEALER on broker1. In this case, it would be broker2 who would maintain a counter to know how many pending requests exist at each moment. However, this solution might require more messages to handle each interaction between clients and workers .

As can be seen, there are multiple alternatives. In all of them, the identity of the client or the worker is retransmitted , obtained upon receiving the corresponding message in the frontend and backend sockets of broker1 and broker2 to correctly maintain which requests are pending and which client originated each of them, or which workers they are available when there are no pending requests.

Solutions based on REQ and REP sockets, although they might work, would not be recommended, as they would unnecessarily block the transfer of information between broker1 and broker2. For example, the transfer of two requests to multiple workers would be complicated if the operations to be executed were long: the second request could not be transmitted to broker2 until the first response had reached broker1.