TSR – Examen de la Práctica 2

No olvide escribir su nombre, apellidos y **GRUPO DE PRÁCTICAS AL QUE PERTENECE** en cada una de las hojas de respuesta que utilice. En algunas cuestiones (p.ej., segunda) podría ser necesario modificar o ampliar el código presentado en el enunciado. Puede utilizar los números para indicar qué líneas van a ser reemplazadas o entre qué líneas se añadirán nuevos fragmentos.

CUESTIÓN 1 (3 puntos)

En el último escenario descrito en la primera sesión de la práctica 2 para el patrón de comunicación PUSH-PULL se utilizan tres programas complementarios:

- origen2.js, que conecta su socket PUSH a dos sockets PULL, mantenidos cada uno por un "filtro" diferente. Tras realizar esas conexiones, mediante un bucle, envía cuatro mensajes **(sin esperar nada)** utilizando ese socket PUSH.
- filtro.js, que realiza un bind de su socket PULL y conecta su socket PUSH al socket PULL del programa "destino". El último argumento que recibe desde la línea de órdenes especifica la pausa, en segundos, tras la que reenviará cada mensaje recibido en su socket PULL por su socket PUSH.
- destino.js, que realiza un bind en su único socket PULL.

En ese escenario (donde no se asume ningún cambio respecto a lo visto en la sesión de laboratorio), se ejecutan estas órdenes:

- terminal 1) **node origen2.js A localhost 9000 localhost 9001**
- terminal 2) **node filtro.js B 9000 localhost 9002 2**
- terminal 3) **node filtro.js C 9001 localhost 9002 3**
- terminal 4) **node destino.js D 9002**

Describa qué pasaría en este ejemplo si el usuario olvidara la orden del terminal 3, lanzando todas las demás (aproximadamente) a la vez. Para ello, responda las siguientes cuestiones:

a) ¿Cuántos mensajes llegarían al proceso "destino"? Justifique por qué.

b) ¿En qué instante llegará (tomando como origen el instante de inicio de "origen2") cada uno de esos mensajes a "destino"?

a) Llegarían la mitad de los mensajes emitidos por "origen2". Obsérvese que "origen2" utiliza un solo socket PUSH para enviar sus mensajes y ha realizado dos connect() antes de enviar su primer mensaje. Los sockets PUSH (al igual que el REQ o el DEALER) utilizar una política de envío circular. Así, el primer mensaje se enviará por la primera conexión, el segundo por la segunda, el tercero de nuevo por la primera y así sucesivamente. Como no hemos iniciado todavía la orden prevista para el tercer terminal, esos mensajes no llegan a salir del socket, pero están asociados a la segunda conexión establecida por el socket PUSH. Si

en algún momento se llegara a ejecutar esa orden, los mensajes pares que debían utilizar esa conexión serían entonces transmitidos hacia el segundo filtro.

b) El programa “filtro.js” realiza una pausa de dos segundos (véase la orden asociada al segundo terminal) antes de reenviar hacia “destino.js” cada mensaje recibido. A su vez, “origen2.js” no realizó ninguna pausa o espera en su bucle de envío, por lo que cada instrucción send() devolverá el control de inmediato y el bucle finalizará en un intervalo muy breve (difícilmente empleará una centésima de segundo en esa tarea, por modesto que sea el ordenador utilizado y alto el número de mensajes a enviar). El patrón de comunicación PUSH-PULL es unidireccional, persistente y asincrónico. Los dos mensajes enviados a través del filtro B serán recibidos por B casi a la vez. El envío de cada uno de ellos se dejará mediante el setTimeout correspondiente al final de la cola de eventos al cabo de dos segundos tras su recepción. Tras esos envíos, esos mensajes necesitarán un intervalo muy breve (pues ambos procesos, emisor y receptor, están en la misma máquina) para llegar a “destino”. Por ello, ambos mensajes llegarán en muy poco más de dos segundos (con una diferencia probablemente inferior a unos pocos milisegundos entre ambas recepciones) al proceso destino D.

CUESTIÓN 2 (4 puntos)

En la práctica 2 se proporcionaba la implementación de un bróker en *nodejs* empleando 2 sockets de tipo router. Dada la implementación original que se incluye a continuación, se pide implementar un nuevo cliente que obtenga ciertas estadísticas del bróker, así como las modificaciones necesarias del bróker original para obtener y proporcionar dichas estadísticas al nuevo cliente. La información a proporcionar es la misma que se pedía en el desarrollo de la práctica 2: número total de peticiones respondidas por el bróker y el número de peticiones respondidas por cada worker que haya atendido al menos una petición en algún momento.

Como aspectos a tener en cuenta:

- El bróker debe atender las peticiones del nuevo cliente mediante un nuevo puerto.
- Para simplificar el código, se admite que todas las estadísticas estén incluidas en una sola cadena de texto, que el bróker enviará al nuevo cliente.

```
1:  const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
2:  lineaOrdenes("frontendPort backendPort")
3:  let workers  =[] // workers disponibles
4:  let pendiente=[] // peticiones no enviadas a ningún worker
5:  let frontend = zmq.socket('router')
6:  let backend  = zmq.socket('router')
7:  creaPuntoConexion(frontend, frontendPort)
8:  creaPuntoConexion(backend,  backendPort)
9:
10: function procesaPeticion(cliente,sep,msg) {
11:         traza('procesaPeticion','cliente sep msg',[cliente,sep,msg])
12:         if (workers.length) backend.send([workers.shift(),'',cliente,'',msg])
13:         else pendiente.push([cliente,msg])
14: }
15: function procesaMsgWorker(worker,sep1,cliente,sep2,resp) {
16:         traza('procesaMsgWorker','worker sep1 cliente sep2 resp',
17:                 [worker, sep1, cliente, sep2, resp])
18:         if (pendiente.length) {
19:                 let [c,m] = pendiente.shift()
20:                 backend.send([worker,'',c,'',m])
21:         }
22:         else workers.push(worker)
23:         if (cliente) frontend.send([cliente,'',resp])
24: }
25: frontend.on('message', procesaPeticion)
26: frontend.on('error'  , (msg) => {error(`${msg}`)})
27:  backend.on('message', procesaMsgWorker)
28:  backend.on('error'  , (msg) => {error(`${msg}`)})
29:  process.on('SIGINT' , adios([frontend, backend],"abortado con CTRL-C"))
```

Los cambios a realizar en el broker serían los siguientes:

a) Aceptar otro argumento desde la línea de órdenes, que podría llamarse statsPort. Para hacerlo, habría que extender la línea 2 y dejarla como sigue:
lineaOrdenes("fontendPort backendPort statsPort")

b) Añadir las siguientes líneas (p.ej., entre la 4 y la 5), para crear el vector de contadores de peticiones procesadas por cada trabajador y el contador global del broker:
let stats = {}
let processed = 0

c) Añadir (p.ej., tras la línea 6) la declaración del nuevo socket. Ese socket debe ser bidireccional. Su tipo podría ser tanto ROUTER como REP. Optamos por esta última alternativa, pues permite simplificar la descripción y desarrollo del cliente:
let statsSocket = zmq.socket("rep")
El nombre de ese socket también debería añadirs al vector pasado como primer argumento en la llamada a adios de la línea 29, pero no es algo crítico.

d) Añadir (p.ej., tras la línea 8) la creación del punto de conexión para el nuevo socket.
creaPuntoConexion(statsSocket, statsPort)

e) Sustituir la línea 23 original por este bloque (o cualquier otro con una funcionalidad similar):

```
if (cliente+"" != "") {
        frontend.send([cliente,'',message])
        processed++
        if (!stats[worker+""])
                stats[worker+""]=1
        else stats[worker+""]++
}
```

f) Añadir (p.ej., entre las líneas 27 y 28) la gestión del evento “message” en el socket statsSocket:

```
statsSocket.on('message', (m)=>{
        let response = "Total requests: "+processed+"\n"
        for (let i in stats)
                response += "Worker " + i +": "+stats[i] + "\n"
        statsSocket.send(response)
})
```

El nuevo cliente sería una copia idéntica del cliente original. No hace falta modificar nada en él para que se comporte como es debido. La única diferencia será, a la hora de iniciarlo, el número de puerto a utilizar como último argumento. En su caso, tendrá que ser el nuevo puerto en el que escuche las peticiones de estadísticas el broker ampliado.

A título ilustrativo, se copia aquí el código del cliente original, pero bastaba con haberlo argumentado en la respuesta proporcionada:

```
const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
lineaOrdenes("id brokerHost brokerPort")
let req = zmq.socket('req')
req.identity = id
conecta(req, brokerHost, brokerPort)
req.send(id)
function procesaRespuesta(msg) {
        adios([req], `Recibido: ${msg}. Adios`)()
}
req.on('message', procesaRespuesta)
req.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([req],"abortado con CTRL-C"))
```

CUESTIÓN 3 (3 puntos)

A continuación, se presenta el código del broker tolerante a fallos, utilizado en la Práctica 3. Explique qué efecto tendría en el funcionamiento de este bróker tolerante a fallos la eliminación de la línea:

if (failed**[**worker**]) return**

dejando el resto del código exactamente igual al código original.

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
const ans_interval = 2000
lineaOrdenes("frontendPort backendPort")
let failed   = {}; let working  = {}; let ready    = []; let pending  = []
let frontend = zmq.socket('router')
let backend  = zmq.socket('router')
function dispatch(client, message) {
        if (ready.length) new_task(ready.shift(), client, message)
        else                        pending.push([client,message])
}
function new_task(worker, client, message) {
        working[worker] = setTimeout(()=>{failure(worker,client,message)}, ans_interval)
        backend.send([worker,'', client,'', message])
}
function failure(worker, client, message) {
        failed[worker] = true;   dispatch(client, message)
}
function frontend_message(client, sep, message) {
        dispatch(client, message)
}
function backend_message(worker, sep1, client, sep2, message) {
        if (failed[worker]) return
        if (worker in working) {
                clearTimeout(working[worker]);  delete(working[worker])
        }
        if (pending.length) new_task(worker, ...pending.shift())
        else ready.push(worker)
        if (client) frontend.send([client,'',message])
}
frontend.on('message', frontend_message)
 backend.on('message', backend_message)
frontend.on('error'  , (msg) => {error(`${msg}`)})
 backend.on('error'  , (msg) => {error(`${msg}`)})
 process.on('SIGINT' , adios([frontend, backend],"abortado con CTRL-C"))
creaPuntoConexion(frontend, frontendPort)
creaPuntoConexion( backend,  backendPort)
```

La línea indicada se utilizaba para evitar que las respuestas tardías generadas por workers a los que se había dado por caídos (por invertir más de dos segundos en contestar) puedan llegar al cliente y esos workers puedan procesar nuevas peticiones o ser incluidos en la cola **ready**. Así se conseguía que esos workers "caídos" pudieran reintegrarse en el sistema.

Si eliminamos esa línea pasarán varias cosas **cuando se reciba una respuesta desde un worker una vez hayan transcurrido dos segundos desde el momento en que se les pasó la última petición dirigida a ellos:**

a) Si hubiera alguna petición pendiente, se reenviaría hacia ese worker lento.

b) Si no, ese worker volvería a la cola **ready** y podría atender futuras peticiones de los clientes.

c) En cualquier caso, su respuesta tardía llegaría al cliente que envió la solicitud correspondiente tiempo atrás. Esa misma solicitud, cuando venció el plazo de dos segundos, se reenvió al primer trabajador disponible (si había alguno en **ready**) o quedó al final de la cola **pending**. Por ello, esa petición habrá podido ser atendida ya por otro trabajador y **el cliente podría llegar a recibir dos respuestas (no exactamente, pues la segunda no sería entregada mientras el cliente no envíe alguna nueva petición) para una misma solicitud**. En nuestro sistema no tendrá un efecto grave, pues el cliente emite una sola petición y termina tan pronto como reciba su respuesta. No le importa que venga de un trabajador o de otro o que en un futuro pudiera recibir otra respuesta.

   Sin embargo, si en lugar de emitir una sola solicitud, el cliente pudiera generar varias de manera secuencial, la llegada de dos respuestas para una misma solicitud habría podido tener efectos graves. Por ejemplo, si el cliente realiza una secuencia de solicitudes A, B y C con diferentes argumentos, esperando en cada una de ellas una respuesta diferente, si el primer trabajador W1 que recibió A respondiera tarde, su respuesta a A quizá llegue cuando el cliente ya hubiera enviado B (pues otro trabajador W2 reemplazó a W1 y envió dentro de plazo la respuesta prevista para A) y sería tomada como la respuesta a B. A su vez, cuando W3 (que recibió B) responda al cliente, su respuesta podría interpretarse como asociada a C. Con ello, todas las solicitudes posteriores a la que fue atendida por el trabajador "lento" recibirían respuestas incorrectas y el servicio proporcionado sería erróneo para todas ellas.