TSR - Recuperación de la práctica 2

No olvides escribir tu nombre, apellidos y **GRUPO DE LABORATORIO AL QUE PERTENECES** en cada una de las hojas de respuesta. En algunas preguntas (por ejemplo, la segunda) puede ser necesario modificar o ampliar el código presentado en esa pregunta. Puedes utilizar los números para indicar qué líneas deben reemplazarse o entre qué líneas se agregarán nuevos fragmentos.

PREGUNTA 1 (5 puntos)

En la sesión 3, el broker debe dividirse en dos partes: un broker que interactúa con los clientes (BC) y un broker que interactúa con los workers (BW). Esos brokers también deben interactuar entre sí. Explique **al menos dos enfoques** para implementar correctamente (es decir, los mensajes previstos deben entregarse adecuadamente) esa comunicación entre brokers, discutiendo las ventajas e inconvenientes de cada enfoque, si los hubiere. A tal efecto, indique claramente, **para cada enfoque** :

1. Qué socket o sockets se deben utilizar en cada broker.
2. Si la comunicación entre brokers es sincrónica o asincrónica.
3. Si alguna parte (ya sea BC o BW) debe iniciar la comunicación o ambas pueden iniciarla libremente.
4. Ventajas o inconvenientes de cada enfoque en comparación con el otro.

Hay un número alto de soluciones admisibles en esta cuestión, por lo que agrupar esos enfoques en parejas y comparar cada combinación generaría una descripción excesivamente extensa. Por ello, discutiremos únicamente los principios generales a considerar.
Para que un enfoque de implementación sea correcto debe permitir que todos los mensajes se entreguen en todos los casos. Si tenemos el broker dividido en dos mitades no debería suceder que, por ejemplo, en un escenario inicial empezaran antes los workers y no pudiéramos reflejar eso correctamente en el broker dividido, por no poder enviar los mensajes entre BC y BW si así debiera suceder.
Como la comunicación entre BC y BW será bidireccional, para que no haya problemas en los envíos entre esas dos mitades también interesaría que esa comunicación fuera asincrónica. Por tanto, parece conveniente utilizar algún patrón de comunicación que sea asincrónico. Eso decartaría el patrón REQ-REP o nos obligaría a explicar con detenimiento de qué manera conseguiríamos transmitir todos los mensajes que deban intercambiar BC y BW si usáramos ese patrón y cuántos sockets necesitaríamos. Habría que razonar que no se bloqueará el envío de ningún mensaje indefinidamente. Sería difícil hacerlo.
Para lograr la bidireccionalidad podemos utilizar un patrón bidireccional y usar un socket en cada parte del broker. Otra opción es emplear un patrón unidireccional, pero establecer dos canales de comunicación entre esas dos partes. Ambas opciones serían válidas.
El tercer punto a considerar (si ambas partes podrán iniciar la comunicación o solo podría hacerlo una de ellas) guarda relación con los sockets que hayamos seleccionado. Debemos observar que un socket ROUTER no puede iniciar la comunicación, pues a la hora de realizar un envío necesita utilizar en su primer segmento la identidad del otro socket y solo la conocerá cuando haya recibido un primer mensaje desde él. Algo similar ocurre con un socket REP, pues no podrá enviar un mensaje (sus mensajes siempre se consideran respuestas) mientras no haya recibido un mensaje previo desde el otro socket.
Las ventajas e inconvenientes de cada enfoque dependerán del otro enfoque con el que se compare. Por ejemplo, comparar un patrón bidireccional DEALER-ROUTER frente a otro enfoque basado en dos canales unidireccionales PUSH-PULL ofrecería como ventaja para el DEALER-ROUTER el uso de un solo socket en cada proceso, pero como inconveniente que

aquella mitad que use el ROUTER no podrá iniciar la comunicación, mientras que con un doble canal PUSH-PULL ambas partes podrían iniciarla libremente.

PREGUNTA 2 (5 puntos)

En el sistema CBW, los mensajes enviados por el broker a los workers incluyen un segmento con la identidad del cliente solicitante. Esa información la recibe el worker y se incluye en su respuesta al broker. Por tanto, un worker malintencionado podría alterar ese identificador.

Por favor, modifique los programas del broker y del worker (y, quizás, la estructura de los mensajes intercambiados) para evitar que el worker pueda manejar la identidad del cliente. No cambie los tipos de sockets.

PROGRAMA DEL WORKER

```
 1: const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
 2: lineaOrdenes("id brokerHost brokerPort")
 3: let req = zmq.socket('req')
 4: req.identity = id
 5: conecta(req, brokerHost, brokerPort)
 6: req.send(['','',''])
 7:
 8: function procesaPeticion(cliente, separador, mensaje) {
 9:         traza('procesaPeticion','cliente separador mensaje',
10:                 [cliente, separador, mensaje])
11:         setTimeout(()=>{req.send([cliente,'',`${mensaje} ${id}`])}, 1000)
12: }
13: req.on('message', procesaPeticion)
14: req.on('error', (msg) => {error(`${msg}`)})
15: process.on('SIGINT', adios([req],"abortado con CTRL-C"))
```

PROGRAMA DEL BROKER

```
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"))
```

Hay diferentes maneras de desarrollar una solución para este ejercicio. Se va a proporcionar una descripción general.
Debemos observar que en todo momento, debido a los sockets que utilizan (REQ en ambos casos), un cliente solo puede tener una petición siendo atendida por los demás componentes, mientras que un trabajador solo puede estar procesando una única petición (entre otras cosas, porque el broker no le habrá reenviado ninguna otra). Por ello, si ahora mismo el trabajador W1 está atendiendo la última petición del cliente C3, el broker podría haberse anotado en alguna estructura de datos a qué trabajador envió esa petición de ese cliente. Con ello, al recibir la respuesta de ese trabajador, consultaría esa estructura de datos y sabría que tendría que redirigirla hacia C3. De esa manera no sería necesario la inclusión del identificador del cliente en los mensajes a recibir por los trabajadores.
Como ya se ha comentado, pueden utilizarse diferentes estructuras de datos con este objetivo (vector o vectores, propiedades de un objeto...) y todas ellas podrían cubrir los objetivos propuestos.

A título de ejemplo, se describe seguidamente una de las posibles formas de desarrollar esa solución. En ella solo se necesita aplicar cambios en el programa del broker. Serían estos:

- Declarar un nuevo vector w2c en la parte inicial del programa. Por ejemplo, entre las líneas 4 y 5. En él dejaríamos la identidad del cliente que haya generado la última petición reenviada al worker de cada una de las componentes de w2c.

```
let w2c = [ ]
```

- Sustituir la línea 12 por este bloque:

```
if (workers.length) {
        w = workers.shift()
        backend.send([w,'','4','',msg])
        w2c[w]=cliente
}
```

  De esta manera no le pasamos ningún identificador real de cliente al worker, sino una cadena fija. Si un trabajador malintencionado quisiera modificar esa información, el cambio no tendrá ningún efecto, pues no utilizaremos esa información al procesar la respuesta. Además guardamos la identidad del cliente en la componente adecuada del vector w2c.

- Sustituir la línea 20 por este bloque:

```
backend.send([worker,'', '4','',m])
w2c[worker]=c
```

  El efecto de este cambio es idéntico al del anterior. Entonces se aplicó en caso de que llegara una solicitud del cliente y hubiese workers esperando, ahora se aplica cuando estamos procesando una respuesta enviada desde un trabajador y había alguna solicitud de los clientes guardada en el vector “pendiente”.

- Sustituir la línea 23 por esta:

```
if (w2c[worker]) frontend.send([w2c[worker],'',resp])
```

  Obsérvese que aquí, en lugar de comprobar si el segmento que mantiene la identidad del cliente (en el mensaje de respuesta enviado por el worker) no está vacío, lo que hacemos es revisar si la componente correspondiente en w2c para ese worker no está vacía. Si no lo está, guardará la identidad del cliente al que hay que reenviar esa respuesta y se reenvía a él.

Otra solución, algo más breve, consistiría en no facilitar ningún segmento con la identidad del cliente en los mensajes que el broker envíe y reciba del worker. En ese caso, también se necesitaría modificar ligeramente el worker, eliminando los segmentos correspondientes. En cualquier caso, esta segunda solución también utilizaría el vector w2c descrito en la solución anterior, pero reduciendo el contenido de los mensajes intercambiados entre broker y worker. Está disponible en el ZIP adjunto.