TSR, grupo 4GIA. Soluciones laboratorio 2 (13/12/2023)

Esta prueba consta de 3 preguntas. Las 2 primeras referencian el siguiente código del ***broker router-router*** *tolerante a fallos de los workers*.

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
const ans_interval = 2000 // deadline to detect worker failure
lineaOrdenes("frontendPort backendPort")

let failed   = {}   // Map(worker:bool) failed workers has an entry
let working  = {}   // Map(worker:timeout) timeouts for workers executing tasks
let ready    = []   // List(worker) ready workers (for load-balance)
let pending  = []   // List([client,message]) requests waiting for workers
let frontend = zmq.socket('router')
let backend  = zmq.socket('router')

function dispatch(client, message) {
    traza('dispatch','client message',[client,message])
    if (ready.length) new_task(ready.shift(), client, message)
    else              pending.push([client,message])
}
function new_task(worker, client, message) {
    traza('new_task','client message',[client,message])
    working[worker] = setTimeout(()=>{failure(worker,client,message)}, ans_interval)
    backend.send([worker,'', client,'', message])
}
function failure(worker, client, message) {
    traza('failure','client message',[client,message])
    failed[worker] = true
    dispatch(client, message)
}
function frontend_message(client, sep, message) {
    traza('frontend_message','client sep message',[client,sep,message])
    dispatch(client, message)
}
function backend_message(worker, sep1, client, sep2, message) {
    traza('backend_message','worker sep1 client sep2 message',[worker,sep1,client,sep2,message])
    if (failed[worker]) return  // ignore messages from failed nodes
    if (worker in working) { // task response in-time
        clearTimeout(working[worker]) // cancel timeout
        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)
```

Pregunta 1 (3 puntos).

Se desea incorporar al `broker` anterior la capacidad de informar sobre **el número de peticiones de clientes pendientes** (`PePe`) de finalizar. Para ello modificarás su código de manera que pueda calcular dicho valor `PePe`, y devolverlo como resultado de las nuevas solicitudes en las que el tipo de socket, puerto y conexión **debe ser compatible** con el siguiente código que representa a un solicitante:

```
// componente que consulta Peticiones Pendientes del `broker`

const {zmq, conecta} = require('../tsr')
let s = zmq.socket('req')
conecta(s, "localhost", 3333)

s.on('message', (PePe) => {
  console.log("Quedan "+PePe+" peticiones pendientes de finalizar")
})

setInterval(()=>{s.send("stats")}, 1000) // el contenido del mensaje no importa
```

Solución pregunta 1.

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('
../tsr')
const ans_interval = 2000 // deadline to detect worker failure
lineaOrdenes("frontendPort backendPort")

let failed   = {}   // Map(worker:bool) failed workers has an entry
let working  = {}   // Map(worker:timeout) timeouts for workers executing tasks
let ready    = []   // List(worker) ready workers (for load-balance)
let pending  = []   // List([client,message]) requests waiting for workers
let frontend = zmq.socket('router')
let backend  = zmq.socket('router')
let PePe     = 0 ①
let p        = zmq.socket('rep') ②

function dispatch(client, message) {
```

```
15     traza('dispatch','client message',[client,message])
16     if (ready.length) new_task(ready.shift(), client, message)
17     else              pending.push([client,message])
18 }
19 function new_task(worker, client, message) {
20     traza('new_task','client message',[client,message])
21     working[worker] = setTimeout(()=>{failure(worker,client,message)},
   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     ++PePe ④
32     dispatch(client, message)
33 }
34 function backend_message(worker, sep1, client, sep2, message) {
35     traza('backend_message','worker sep1 client sep2 message',[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]); --PePe} ⑤
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  p.on('message', (dummy) => {p.send(PePe)}) ⑥
53
54 creaPuntoConexion(frontend, frontendPort)
55 creaPuntoConexion( backend,  backendPort)
56 creaPuntoConexion(p, 3333) ③
```

① Un contador `PePe` para las peticiones pendientes, inicializado a `0`

② Un socket `p` de tipo `rep`, que puede intercambiar mensajes con el socket `req` del componente que realiza la consulta

③ Conexión `bind` que encaja con `connect` del componente que realiza la consulta, puerto `3333`

④ Incrementar `PePe` con la llegada de una nueva petición, de momento sin finalizar

⑤ Decrementar `PePe` con la comunicación del resultado al cliente

⑥ Independientemente del contenido del mensaje recibido (`dummy`), devolver el valor de `PePe`

Soluciones alternativas

Una reflexión acertada sería pensar que el valor devuelto incluye tanto las peticiones en proceso (asignadas a un trabajador) como las encoladas (en pendientes, a la espera de un trabajador disponible). Como el primer elemento coincide con `working.length`, y el segundo con `pending.length`, el valor a devolver será `working.length + pending.length`.

Pregunta 2 (5 puntos).

Partiendo del `broker` tolerante a fallos incluido en este enunciado, se solicita añadir un nuevo segmento a cada respuesta devuelta a cada cliente. El contenido de dicho segmento será un número que represente en **cuántas ocasiones se ha reasignado la petición a un nuevo `worker`, como consecuencia de fallos en trabajadores**.

- Una petición en cuya atención no se haya detectado ningún fallo de trabajador incluirá un **0** (cero) como valor adicional del mensaje de respuesta.
- Si en la atención de una solicitud de un cliente se detectara un fallo del trabajador encargado, ese valor adicional devuelto al cliente será **1**.

En general, si se ha necesitado reenviar la petición a otros **N** trabajadores (sin incluir el trabajador inicial), ése será el valor **N** que debe devolverse como nuevo segmento al cliente.

Solución pregunta 2.

Nuestra primera preocupación es idear cómo mantener la información de cada petición. Podemos optar por una estructura, tipo diccionario, como la empleada para `working`, si encontramos un criterio para identificar y acceder a cada elemento.

- La estructura `working` cuenta con el identificador del trabajador para referenciar su contenido, pero, en nuestro caso, no existe un identificador de petición que podamos aplicar (y parece complicado inventarnos algo de ese tipo).
- Sin embargo no es difícil encontrar algo equivalente si nos fijamos en un detalle relevante: como las solicitudes de los clientes se envían mediante un socket `req`, no es posible que haya más de una petición pendiente por cliente.

De aquí podemos deducir que el identificador de cliente es un candidato ideal para ser empleado como índice para acceder a la información de su, potencialmente, única petición pendiente.

Si estudiamos el *ciclo de vida* de `working` podemos tomarlo como referencia para nuestra nueva estructura: declaración (línea 6 original), añadir entrada (línea 19 original), comprobar existencia (línea 34 original), eliminar (línea 36 original, pero no lo usamos porque no necesitamos comprobar la existencia de la entrada).

Llamamos a nuestra estructura de datos `reasig`.

```
const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
const ans_interval = 2000 // deadline to detect worker failure
lineaOrdenes("frontendPort backendPort")

let failed   = {}   // Map(worker:bool) failed workers has an entry
let working  = {}   // Map(worker:timeout) timeouts for workers executing tasks
let reasig   = {}   // Map(client:counter) counters for request retries ①
let ready    = []   // List(worker) ready workers (for load-balance)
let pending  = []   // List([client,message]) requests waiting for workers
let frontend = zmq.socket('router')
let backend  = zmq.socket('router')

function dispatch(client, message) {
    traza('dispatch','client message',[client,message])
    if (ready.length) new_task(ready.shift(), client, message)
    else              pending.push([client,message])
}
function new_task(worker, client, message) {
    traza('new_task','client message',[client,message])
    working[worker] = setTimeout(()=>{failure(worker,client,message)}, ans_interval)
    backend.send([worker,'', client,'', message])
}
function failure(worker, client, message) {
    traza('failure','client message',[client,message])
    ++reasig[client] ③
    failed[worker] = true
    dispatch(client, message)
}
function frontend_message(client, sep, message) {
    traza('frontend_message','client sep message',[client,sep,message])
    reasig[client]=0 ②
    dispatch(client, message)
}
function backend_message(worker, sep1, client, sep2, message) {
    traza('backend_message','worker sep1 client sep2 message',[worker,sep1,client,sep2,message])
    if (failed[worker]) return  // ignore messages from failed nodes
    if (worker in working) { // task response in-time
        clearTimeout(working[worker]) // cancel timeout
        delete(working[worker])
    }
    if (pending.length) new_task(worker, ...pending.shift())
    else ready.push(worker)
    if (client != "") frontend.send([client,'',message,reasig[client]]) ④
}

frontend.on('message', frontend_message)
 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 Declaración de la estructura `reasig`.

② La llegada de una nueva solicitud de un cliente `c` ⇒ inicializar `reasig[c]`

③ Notificación de un fallo por *timeout* (función `failure`) ⇒ incrementar `reasig[c]`

④ Devolver un resultado al cliente `c` ⇒ añadir segmento con `reasig[c]`

Soluciones alternativas

En lugar de una información centralizada (estructura `reasig`), podemos añadir un contador (inicializado a 0) como último segmento de la petición procedente del cliente. En cada ocasión en que se ejecute `failure`, ese contador se incrementará antes de pasar la petición a un nuevo trabajador, y formará parte de la respuesta devuelta al cliente. El mayor inconveniente que posee es que deben cambiarse todas las cabeceras de función que dependen del número de piezas del mensaje.

Pregunta 3 (2 puntos).

En el apartado de la práctica 2 relativo al patrón pipeline (`push/pull`), en la combinación **`origen1`** ⇒ **`filtro`** ⇒ **`destino A-B-C`** se solicita la ejecución de órdenes en 3 terminales:

```
terminal 1) node origen1.js A localhost 9000
terminal 2) node filtro.js B 9000 localhost 9001 2
terminal 3) node destino.js C 9001
```

El código de los tres componentes mencionados es:

`origen1.js`

```
1  const {zmq, lineaOrdenes, error, adios, conecta} = require('../tsr')
2  lineaOrdenes("nombre hostSig portSig")
3
4  let salida = zmq.socket('push')
5  conecta(salida, hostSig, portSig)
6
7  salida.on('error', (msg) => {error(`${msg}`)})
8  process.on('SIGINT', adios([salida],"abortado con CTRL-C"))
9
10 for (let i=1; i<=4; i++) {
11     console.log(`enviando mensaje: [${nombre},${i}]`)
12     salida.send([nombre,i])
13 }
```

destino.js

```
const {zmq, error, lineaOrdenes, traza, adios, creaPuntoConexion} = require('
../tsr')
lineaOrdenes("nombre port")

var entrada = zmq.socket('pull')
creaPuntoConexion(entrada, port)

function procesaMensaje(filtro, nombre, iteracion) {
    if (!iteracion) {
        iteracion = nombre
        nombre = filtro
        filtro = ""
    }
    traza('procesaMensaje','filtro nombre iteracion',[filtro, nombre, iteracion])
}
entrada.on('message', procesaMensaje)
entrada.on('error', (msg) => {error(`${msg}`)})
process.on('SIGINT', adios([entrada],"abortado con CTRL-C"))
```

filtro.js

```
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"))
```

Contesta a estas dos preguntas incluidas en este apartado [1]:

- Indica la razón por la que `origen1.js` y `destino.js` definen un único socket cada uno, pero `filtro.js` define 2 sockets

`filtro.js` necesita un socket para comunicar con `origen1.js` (recibir sus mensajes) y necesita otro socket diferente para comunicar con `destino.js` (enviarle peticiones). En las condiciones de la práctica 2 no hay ningún otro componente con el que comunicarse.

- Si `origen1` genera 4 mensajes y `filtro` retarda 2 segundos, ¿cúanto crees que tarda el último mensaje de `origen1` en llegar a `destino`?

`origen1` genera 4 mensajes y los envía **todos** (uno tras otro) a `filtro` prácticamente sin retardo mediante un socket asíncrono. En resumen: en el instante `0` los 4 mensajes han salido de `origen1` y se encuentran a las puertas de `filtro`

`filtro` recibe cada mensaje, inicia el temporizador adecuado para cada uno y queda disponible para el siguiente, lo que consume un tiempo mínimo. Por tanto, en un momento dado, pueden estar en marcha los 4 temporizadores, con *muy pequeñas diferencias en su vencimiento*. El primero finalizará muy poco después del instante `2`, desencadenando un envío (`salida.send()`) sobre un socket asincrónico que no provoca un retardo apreciable. No es significativo que, mientras tanto, puedan vencer los demás temporizadores; el hecho es que transcurrirá un tiempo muy breve entre esos vencimientos y el último de los envíos (prácticamente finaliza todo en el instante `2`).

El procesamiento que `destino.js` hace para cada mensaje recibido por su socket asincrónico es casi instantáneo, por lo que desde la recepción del primer mensaje hasta el último no se aprecia ningún retardo ya que `destino.js` está disponible para atenderlos.

En conclusión, el tiempo transcurrido entre la recepción del primer y último mensaje de `filtro` es prácticamente `0`. Si lo sumamos al instante `2` que marcaba la salida del último mensaje desde `filtro`, tenemos que el último mensaje de `origen1` tarda *un poco más* de `2` segundos en llegar a `destino`.

Obsérvese que se necesitaría un número de mensajes mucho más elevado para que el tiempo se modificara apreciablemente.

[1] Las respuestas aparecen tras las preguntas