TSR

Examen de la sesión 3 de la Práctica 2 (13 de enero de 2025)

Dado el código del **broker tolerante a fallos** utilizado en la última sesión de la práctica 2:

```
 1: const {zmq,lineaOrdenes,traza,error,adios,creaPuntoConexion} = require('../tsr')
 2: const ans_interval = 2000 // deadline to detect worker failure
 3: lineaOrdenes("frontendPort backendPort")
 4: let failed   = {}  // Map(worker:bool) failed workers has an entry
 5: let working  = {}          // Map(worker:timeout) for workers executing tasks
 6: let ready    = [] // List(worker) ready workers (for load-balance)
 7: let pending  = []          // List([client,message]) requests waiting for workers
 8: let frontend = zmq.socket('router')
 9: let backend  = zmq.socket('router')
10: function dispatch(client, message) {
11:         traza('dispatch','client message',[client,message])
12:         if (ready.length) new_task(ready.shift(), client, message)
13:         else pending.push([client,message])
14: }
15: function new_task(worker, client, msg) {
16:         traza('new_task','client message',[client,msg])
17:         working[worker]=setTimeout(()=>{failure(worker,client,msg)}, ans_interval)
18:         backend.send([worker,'', client,'', msg])
19: }
20: function failure(worker, client, message) {
21:         traza('failure','client message',[client,message])
22:         failed[worker] = true
23:         dispatch(client, message)
24: }
25: function frontend_message(client, sep, message) {
26:         traza('frontend_message','client sep message',[client,sep,message])
27:         dispatch(client, message)
28: }
29: function backend_message(worker, sep1, client, sep2, message) {
30:         traza('backend_message','worker sep1 client sep2 message',
31:                 [worker,sep1,client,sep2,message])
32:         if (failed[worker]) return  // ignore messages from failed nodes
33:         if (worker in working) { // task response in-time
34:                 clearTimeout(working[worker]) // cancel timeout
35:                 delete(working[worker])
36:         }
37:         if (pending.length) new_task(worker, ...pending.shift())
38:         else ready.push(worker)
39:         if (client) frontend.send([client,'',message])
40: }
41: frontend.on('message', frontend_message)
42:  backend.on('message', backend_message)
43: frontend.on('error'  , (msg) => {error(`${msg}`)})
44:  backend.on('error'  , (msg) => {error(`${msg}`)})
45:  process.on('SIGINT' , adios([frontend, backend],"abortado con CTRL-C"))
46: creaPuntoConexion(frontend, frontendPort)
47: creaPuntoConexion( backend,  backendPort)
```

Se pretende **añadir un cuarto componente** al sistema CBW tolerante a fallos: el **operador**. Este componente se encarga de "reparar" los trabajadores que el broker detecte como fallidos. Para ello…

- Cuando el broker detecte un trabajador fallido, enviará un mensaje al operador (además de su reacción normal) con el identificador del trabajador. Tanto el broker como el operador habrán recibido inicialmente desde la línea de órdenes la información necesaria para, creando el socket o sockets pertinentes, permitir que esa comunicación sea posible. **(30%)**

- La "reparación" de un trabajador, por parte del operador, será una actividad EMULADA mediante una espera de 40 segundos. Cuando el operador finalice la reparación de un trabajador, ese trabajador será reiniciado por el operador, con su mismo identificador y de manera automatizada utilizando la función *reiniciar(idWorker)*, que se supone ya implementada y directamente accesible sin necesidad de importar ningún módulo. **(20%)**

- Al ser reiniciado, el trabajador se reconectará al broker, por lo que la adaptación del broker debe admitir esa reconexión (es decir, aceptar la recepción de un mensaje de registro para un trabajador que había fallado previamente). **(15%)**

- El broker debe seguir rechazando una respuesta de un trabajador lento si se le había dado por caído y todavía no se ha reconectado tras su reparación. **(15%)**

- El operador mostrará por pantalla la relación de trabajadores en reparación (puede haber varios en esa situación) cada 5 segundos. Obsérvese que los trabajadores ya reparados no deben aparecer en esa relación. **(20%)**

Se solicita **extender el código del broker tolerante a fallos, así como desarrollar el programa operador.js** para implantar el sistema descrito. No se necesitará aplicar ningún cambio en el programa workerReq.js ni en el programa utilizado por los clientes.

Las modificaciones a aplicar en el broker pueden especificarse indicando entre qué líneas habría que insertar qué código y qué otras líneas habría que eliminar o sustituir.

Recuérdese que el módulo `tsr.js` también ofrece una operación `conecta(socket,ip,port)` que podría resultar necesaria en las extensiones del broker o en el código del programa operador.js.

SOLUCIÓN

No existe una única solución para este ejercicio, pues el enunciado admite diferentes formas de resolver las condiciones exigidas. Una de ellas, en lo que afecta al broker, se presenta seguidamente manteniendo la numeración original de cada línea para resaltar las ampliaciones aplicadas:

```
 1: const {zmq, lineaOrdenes, traza, error, adios, creaPuntoConexion} = require('../tsr')
 2: const ans_interval = 2000 // deadline to detect worker failure
 3: lineaOrdenes("frontendPort backendPort operatorPort")
 4: let failed   = {}        // Map(worker:bool) failed workers has an entry
 5: let working  = {}        // Map(worker:timeout) timeouts for workers executing tasks
 6: let ready    = []        // List(worker) ready workers (for load-balance)
 7: let pending  = []        // List([client,message]) requests waiting for workers
 8: let frontend = zmq.socket('router')
 9: let backend  = zmq.socket('router')
    let operator = zmq.socket('push')
10: function dispatch(client, message) {
11:             traza('dispatch','client message',[client,message])
12:             if (ready.length) new_task(ready.shift(), client, message)
13:             else  pending.push([client,message])
14: }
15: function new_task(worker, client, message) {
16:             traza('new_task','client message',[client,message])
17:             working[worker] = setTimeout(()=>{failure(worker,client,message)}, ans_interval)
18:             backend.send([worker,'', client,'', message])
19: }
20: function failure(worker, client, message) {
21:             traza('failure','client message',[client,message])
22:             failed[worker] = true
                delete(working[worker])
                operator.send(worker+"")
23:             dispatch(client, message)
24: }
25: function frontend_message(client, sep, message) {
26:             traza('frontend_message','client sep message',[client,sep,message])
27:             dispatch(client, message)
28: }
29: function backend_message(worker, sep1, client, sep2, message) {
30:             traza('backend_message','worker sep1 client sep2 message',
31:                     [worker,sep1,client,sep2,message])
32:             if (failed[worker] && client!="") return  // ignore responses from failed nodes
                if (failed[worker] && client=="") failed[worker]=false //accept again a repaired worker
33:             if (worker in working) { // task response in-time
34:                     clearTimeout(working[worker]) // cancel timeout
35:                     delete(working[worker])
36:             }
37:             if (pending.length) new_task(worker, ...pending.shift())
38:             else ready.push(worker)
39:             if (client) frontend.send([client,'',message])
40: }
41: frontend.on('message', frontend_message)
42:  backend.on('message', backend_message)
43: frontend.on('error'  , (msg) => {error(`${msg}`)})
44:  backend.on('error'  , (msg) => {error(`${msg}`)})
    operator.on('error'  , (msg) => {error(`${msg}`)})
45:  process.on('SIGINT' , adios([frontend, backend],"abortado con CTRL-C"))
46: creaPuntoConexion(frontend, frontendPort)
47: creaPuntoConexion( backend,  backendPort)
    creaPuntoConexion(operator, operatorPort)
```

Como puede observarse, se ha añadido en la línea 3 original el puerto en el que esperar conexiones desde el operador. Se asume que el broker está realizando la operación creaPuntoConexion sobre ese puerto, pero también se aceptaría que fuera el operador quien realizase esa acción y que el broker recibiese tanto la dirección IP como el puerto del operador y se conectara a este último. Ambas opciones son válidas, pues solo habrá un operador y un broker en un mismo sistema, por lo que nada condiciona la elección entre las operaciones bind y connect subyacentes.

Entre las líneas 9 y 10 se ha insertado otra para crear un socket de tipo PUSH, pues en esta solución se está asumiendo que una vez reiniciado, el propio trabajador enviará un nuevo mensaje de registro (con todos sus segmentos vacíos) y de esa manera el broker sabrá que ya está recuperado. Por ello, la comunicación entre broker y operador puede ser unidireccional. También se admitiría el haber empleado comunicación bidireccional entre ambos componentes, de manera que el operador respondiera al broker tan pronto como hubiese finalizado la recuperación de un trabajador. En ese hipotético caso, la comunicación debería ser asincrónica, pues se podría estar gestionando un alto número de trabajadores y ello permitiría que fallaran y debieran recuperarse varios en un plazo de cuarenta segundos.

Entre las líneas 22 y 23 se ha enviado al operador la identidad del trabajador que haya fallado. Ese envío puede realizarse en cualquier posición dentro del cuerpo de la función *failure*.

La línea 32 original debía ampliarse para comprobar en su condición que el mensaje recibido desde un trabajador para el que creíamos que había fallado no fuese un mensaje de registro. De esa manera se daba soporte a la condición exigida en el penúltimo punto del enunciado.

Entre las líneas 32 y 33 originales añadimos el código necesario para admitir un nuevo mensaje de registro de algún worker que haya sido reparado y ponemos a **false** su componente en failed para indicar que vuelve a estar activo, por lo que podremos reenviarle las peticiones de los clientes y aceptaremos de nuevo las respuestas que proporcione. Habría bastado con

```
if (failed[worker]) failed[worker]=false
```

pues el hecho de que el segmento *client* sea la cadena vacía está implícito al alcanzar esa línea. Con esta ampliación se cumple la antepenúltima condición mencionada en el enunciado.

Entre las líneas 44 y 45 se añade la gestión de errores de comunicación en el socket que permite interactuar con el operador. Esto no llegaba a solicitarse en el enunciado. Se incluye por completitud pero debe considerarse opcional y sin ningún impacto en la calificación de este ejercicio.

Finalmente, tras la línea 47 se realiza la operación *creaPuntoConexion* sobre el socket que permite interactuar con el operador. Ya hemos discutido sobre esto al describir la ampliación entre las líneas 9 y 10.

Seguidamente se muestra el código necesario para implantar un operador que pueda interactuar con este broker extendido. De nuevo, podrían darse múltiples variantes válidas de este programa.

```
 1: const {zmq, lineaOrdenes, traza, error, adios, conecta} = require('../tsr')
 2: lineaOrdenes("id brokerHost brokerPort")
 3: const repairDelay = 40000
 4: const reportInterval = 5000
 5: let pu = zmq.socket('pull')
 6: conecta(pu, brokerHost, brokerPort)
 7: let repairing = {} // Set with the identifiers of the workers being repaired.
 8: function processRepairRequest(worker) {
 9:         traza('processRepairRequest','worker',[worker])
10:         repairing[worker+""] = true
11:         setTimeout(()=>{reiniciar(worker);delete(repairing[worker+""])}, repairDelay)
12: }
13: function showRepairReport() {
14:         console.log("WORKERS BEING REPAIRED:")
15:         for(let i in repairing) console.log("  "+i)
16: }
17: pu.on('message', processRepairRequest)
18: pu.on('error', (msg) => {error(`${msg}`)})
19: process.on('SIGINT', adios([pu],"abortado con CTRL-C"))
20: setInterval(showRepairReport, reportInterval)
```