TSR

Examen de la sesión 3 de la Práctica 2 (13 Enero 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.