NIST

Practice 2 Session 3 Exam (January 13, 2025)

Given the **fault-tolerant broker code** used in the last session of practice 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)
```

A fourth component should be added to the fault-tolerant CBW system: the **operator**. This component is responsible for "repairing" workers that the broker detects as failed. To this end...

- When the broker detects a failed worker, it sends a message to the operator (in addition to its normal reaction) with the worker's ID. Both the broker and the operator should have received from the command line the necessary information to create the relevant socket or sockets to enable such communication. **(30%)**

- The "repair" of a worker by the operator will be an EMULATED activity by waiting for 40 seconds. When the operator finishes the repair of a worker, that worker will be restarted by the operator, with its same identifier and in an automated way using the *restart(idWorker)* function, which is assumed to be already implemented and directly accessible without the need to import any module. **(20%)**

- When restarted, the worker reconnects to the broker, so the broker adaptation must support that reconnection (i.e. accept the receipt of a registration message for a worker that had previously failed). **(15%)**

- The broker should continue to reject a response from a slow worker if it was reported down and has not yet come back online after being repaired. **(15%)**

- The operator will display on screen the list of workers in repair (there may be several in this situation) every 5 seconds. Note that workers already repaired should not appear in this list. **(20%)**

It is requested **to extend the fault-tolerant broker code and develop the operator.js program** to implement the described system. No changes will be required to the workerReq.js program or to the program used by clients.

The modifications to be applied to the broker can be specified by indicating between which lines which code should be inserted and which other lines should be deleted or replaced.

The `tsr.js` module also provides a `connect(socket,ip,port)` operation that may be needed in broker extensions or the `operator.js` program code.