TSR

Examen de recuperación de la Práctica 2 (30 Enero 2025)

Esta prueba se compone de dos preguntas. Requiere obtener el mínimo indicado en la guía docente (3 sobre 10), y contribuye con 3 puntos a la nota final.

---

1. (**5 puntos**. Contesta en papel separado) Dado el código del **publicador** de la práctica 2:

```
1:  const {zmq, error, lineaOrdenes, traza, adios, creaPuntoConexion} = require('../tsr')
    lineaOrdenes("port tema1 tema2 tema3")
2:  let temas = [tema1,tema2,tema3]
3:  let pub = zmq.socket('pub')
4:  creaPuntoConexion(pub, port)
5:
6:  function envia(tema, numMensaje, ronda) {
7:    traza('envia','tema numMensaje ronda',[tema, numMensaje, ronda])
8:    pub.send([tema, numMensaje, ronda])
9:  }
10: function publica(i) {
11:   return () => {
12:     envia(temas[i%3], i, Math.trunc(i/3))
13:     if (i==10) adios([pub],"No me queda nada que publicar. Adios")()
14:     else setTimeout(publica(i+1),1000)
15:   }
16: }
17: setTimeout(publica(0), 1000)
18: pub.on('error', (msg) => {error(`${msg}`)})
19: process.on('SIGINT', adios([pub],"abortado con CTRL-C"))
```

Modifique este programa de manera que respete todas estas condiciones simultáneamente:

a) Emitirá un mensaje **periódicamente**, alternando cíclicamente entre todos los temas especificados en los argumentos recibidos, sin fin. (**30%**)

b) El número de temas a utilizar lo decidirá el usuario en cada ejecución, facilitando para ello los argumentos necesarios en la línea de órdenes. (**30%**)

c) Los mensajes se difundirán cada medio segundo. (**10%**)

d) No debe utilizarse `setTimeout` y tampoco una variable global para dar el valor de `numMensajes` cuando se invoque la función `envia`. (**30%**)

SOLUCIÓN: Habrá múltiples implementaciones posibles que respetarán todas las condiciones enunciadas. Se presenta seguidamente una de ellas, pero eso no excluye a las demás:

```
1: const {zmq, error, traza, adios, creaPuntoConexion} = require('../tsr')
2: if (process.argv.length < 4) {
3:         console.error("Debes proporcionar un número de puerto y algún tema.")
4:         process.exit(1)
5: }
6: let temas = process.argv.slice(3)
7: const pub = zmq.socket('pub')
8: creaPuntoConexion(pub, parseInt(process.argv[2]))
9: function envia(tema, numMensaje, ronda) {
10:         traza('envia','tema numMensaje ronda',[tema, numMensaje, ronda])
11:         pub.send([tema, numMensaje, ronda])
12: }
13: function publica() {
14:         let i = 0, nt = temas.length
15:         return () => {
16:                 envia(temas[i%nt], i, Math.trunc(i/nt)); i++
17:         }
18: }
19: setInterval(publica(), 500)
20: pub.on('error', (msg) => {error(`${msg}`)})
21: process.on('SIGINT', adios([pub],"abortado con CTRL-C"))
```

La primera condición se cumple al sustituir *setTimeout* por *setInterval*. Al realizar ese cambio, también nos vemos obligados a eliminar el parámetro que recibía la función *publica* en el código original. El uso de la variable *nt* definida en la línea 14 y utilizada en las expresiones de la línea 16 permite desarrollar la gestión circular de los temas facilitados a la hora de difundir los mensajes correspondientes.

La segunda condición exige que no utilicemos la función *lineaOrdenes* del módulo *tsr*, sino que obtengamos todos los argumentos de interés directamente desde el vector *process.argv*. El código necesario está en las líneas dos a ocho del nuevo programa. Resulta conveniente, aunque no era imprescindible, comprobar cuántos argumentos ha facilitado el usuario, tal como se muestra en el bloque de las líneas dos a cinco.

El nuevo periodo de difusión de los mensajes exigido en la tercera condición se soporta utilizando 500 como segundo argumento en el *setInterval* de la línea 19.

Por último, aquello exigido en la cuarta condición implicaba que no se utilizase un contador global a la hora de facilitar el segundo argumento en nuestras llamadas a la función *envia*. Eso se ha soportado declarando la variable *i* en la línea 14 e incrementándola al final de la línea 16. El programa inicial ya utilizaba una clausura para proporcionar el primer argumento en las llamadas a *setTimeout*, por lo que el cambio a aplicar en la solución mantenía una estructura similar.

ALUMNA/O

TURNO LABORATORIO

1. (**5 puntos**. Contesta en la página siguiente) 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)
```

(Las cuestiones están en la página siguiente)

NO OLVIDES ENTREGAR ESTA HOJA

Cierto programador ha analizado el código de ese broker y ha sugerido que permite gestionar adecuadamente los siguientes escenarios:

a) Reenvío de una petición hacia el primer trabajador disponible, pues hay alguno.
b) Encolado de una petición si no hay trabajadores disponibles.
c) Reenvío de una respuesta hacia su cliente.
d) Reenvío de una petición hacia otro trabajador, cuando falla el trabajador inicialmente asignado.
e) Encolado de una petición tras haber fallado el trabajador hacia el que fue reenviada inicialmente, si no hay otros trabajadores disponibles.
f) Descarte de una respuesta tardía enviada por un trabajador excesivamente lento.
g) Admisión de un mensaje inicial de registro enviado por un nuevo trabajador.
h) Llegada, dentro del plazo previsto, de una respuesta emitida por un trabajador.
i) Reenvío de una petición encolada hacia un trabajador que acaba de quedar libre.

Identifique (marcando en la tabla) qué escenario o escenarios, de entre los que acabamos de listar, podría/n ocasionar que las condiciones utilizadas en las siguientes líneas llegaran a cumplirse y se ejecutaran sus instrucciones asociadas:

i. Línea 12: **if (ready.length) new_task(ready.shift(), client, message)**
ii. Línea 32: **if (failed[worker]) return**
iii. Línea 33: **if (worker in working) { … }**
iv. Línea 37: **if (pending.length) new_task(worker, …pending.shift())**
v. Línea 39: **if (client) frontend.send([client,'',message])**

(contesta en esta misma tabla con SÍ o NO en cada celda)

	a	b	c	d	e	f	g	h	i
i	SÍ								
ii						SÍ			
iii								SÍ	
iv									SÍ
v			SÍ						

SOLUCIÓN: Se ha incluido en la tabla anterior. Todas aquellas celdas vacías deberían contener NO. Cada línea se asociaba a un único escenario de entre los listados.

Obsérvese que había tres modelos de examen, utilizando cada uno de ellos una identificación distinta para los escenarios, por lo que el contenido de la tabla será distinto en los otros dos modelos de examen.