PARTE 1 (6 PUNTOS)

A continuación se muestran los programas *mybroker.js, myclient.js* y *myworker.js* que facilitan la base inicial para desarrollar la segunda parte de la segunda práctica. Son idénticos a los presentados en el boletín.

Tomándolos como base, resuelve las 3 actividades propuestas al final.

```
01:  // ROUTER-ROUTER request-reply broker in NodeJS
02:  var zmq      = require('zmq')
03:    , frontend = zmq.socket('router')
04:    , backend  = zmq.socket('router');
05:
06:  var fePortNbr = 8059;
07:  var bePortNbr = 8060;
08:  var workers = [];
09:  var clients = [];
10:
11:  frontend.bindSync('tcp://*:'+fePortNbr);
12:  backend.bindSync('tcp://*:'+bePortNbr);
13:
14:  frontend.on('message', function() {
15:    var args = Array.apply(null, arguments);
16:    if (workers.length > 0) {
17:      var myWorker = workers.shift();
18:      var m = [myWorker,''].concat(args);
19:      backend.send(m);
20:    } else
21:      clients.push( {id: args[0],msg: args.slice(2)});
22:  });
23:
24:  function processPendingClient(workerID) {
25:    if (clients.length>0) {
26:      var nextClient = clients.shift();
27:      var m = [workerID,'',nextClient.id,''].concat(nextClient.msg);
28:      backend.send(m);
29:      return true;
30:    } else
31:      return false;
32:  }
33:
34:  backend.on('message', function() {
35:    var args = Array.apply(null, arguments);
36:    if (args.length == 3) {
37:      if (!processPendingClient(args[0]))
38:      workers.push(args[0]);
39:    } else {
40:      var workerID = args[0];
41:      args = args.slice(2);
42:      frontend.send(args);
43:      if (!processPendingClient(workerID))
44:        workers.push(workerID);
45:    }
46:  });
```

```
01:  // myclient in NodeJS (myclient.js)
02:  var zmq       = require('zmq')
03:    , requester = zmq.socket('req');
04:
05:  var brokerURL = 'tcp://localhost:8059';
06:  var myID = 'NONE';
07:  var myMsg = 'Hello';
08:
09:  if (myID != 'NONE')
10:    requester.identity = myID;
11:  requester.connect(brokerURL);
12:  console.log('Client (%s) connected to %s', myID, brokerURL)
13:
14:  requester.on('message', function(msg) {
15:    console.log('Client (%s) has received reply "%s"', myID, msg.toString());
16:    process.exit(0);
```

```
17:   });
18:   requester.send(myMsg);
```

```
01:   // myworker server in NodeJS (myworker.js)
02:   var zmq = require('zmq')
03:     , responder = zmq.socket('req');
04:
05:   var backendURL = 'tcp://localhost:8060';
06:   var myID = 'NONE';
07:   var connText = 'id';
08:   var replyText = 'World';
09:
10:   if (myID != 'NONE')
11:     responder.identity = myID;
12:   responder.connect(backendURL);
13:   responder.on('message', function(client, delimiter, msg) {
14:     setTimeout(function() {
15:       responder.send([client,'',replyText]);
16:     }, 1000);
17:   });
18:   responder.send(connText);
```

Se pretende modificar esta solución para que el sistema se comporte de esta manera:

Cuando un cliente envíe una petición y no haya trabajadores disponibles en ese momento en la cola **workers[]**, no esperará en ninguna cola. En su lugar, recibe una respuesta especial: **'NoWorkers'**. Al recibir esa respuesta, el cliente programa un reintento en un segundo y llegará a reintentar hasta cinco veces; si esa quinta respuesta todavía es **'NoWorkers'** muestra un mensaje de error al usuario ("**All workers are busy at the moment!**") y termina.

Las dos primeras actividades requieren modificaciones en los programas. Debes especificar qué cambios se necesitan para implantar lo que se proponga. Para ello:

- Especifica claramente qué líneas de los programas originales deben modificarse.
- En caso de añadir más código, indica entre qué dos líneas debe ser insertado.
- Alternativamente, puedes escribir el programa completo.

Se pide realizar estas 3 actividades:

1. (2 puntos) Escribe las modificaciones necesarias en *myclient.js*.
2. (2 puntos) Escribe las modificaciones necesarias en *mybroker.js*.
3. (2 puntos) Supongamos que en una terminal Linux se han escrito las siguientes líneas de órdenes, sin ninguna pausa entre ellas. Explica cuántos clientes mostrarán el mensaje de error ("**All workers are busy at the moment!**"):

```
$ node mybroker&
$ node myworker&
$ node myclient& node myclient& node myclient& node myclient& node myclient& node myclient& node myclient& node myclient&
```

(las 2 últimas líneas se refieren a una única orden para lanzar 8 clientes simultáneamente)

SOLUCIÓN PARTE 1

1. El componente myclient.js necesita una variable global (por ejemplo, attempts) inicializada a cero para conocer el número de intentos que se han realizado hasta el momento. Esto se puede hacer insertando una nueva línea, entre las originales 7 y 8, con este contenido:

```
var attempts = 0;
```

Adicionalmente, el cuerpo del *listener* para los eventos "message" (líneas 15 y 16) debe ser reemplazado con las líneas 20 a 26 de la solución completa:

```
01:  // myclient in NodeJS (myclient.js)
02:  var zmq       = require('zmq')
03:    , requester = zmq.socket('req');
04:
05:  var brokerURL = 'tcp://localhost:8059';
06:  var myID = 'NONE';
07:  var myMsg = 'Hello';
08:  var attempts = 0;
09:
10:  if (myID != 'NONE')
11:    requester.identity = myID;
12:  requester.connect(brokerURL);
13:  console.log('Client (%s) connected to %s', myID, brokerURL)
14:
15:  requester.on('message', function(msg) {
16:    if (msg != 'NoWorkers') {
17:      console.log('Client (%s) has received reply "%s"', myID, msg.toString());
18:      process.exit(0);
19:    }
20:    else {
21:      attempts++;
22:      if (attempts==6) {
23:        console.log("All workers are busy at the moment!");
24:        process.exit(1);
25:      }
26:      setTimeout( function() { requester.send(myMsg) }, 1000);
27:    });
28:  requester.send(myMsg);
```

2. El nuevo broker no necesita la cola de clientes, puesto que en lugar de esperar cuando no hay workers disponibles los clientes ahora reciben una respuesta especial del broker. Esto significa que en la nueva versión de `mybroker.js` las líneas originales 9, 24-33, 37 y 43 pueden eliminarse opcionalmente. Sin embargo, la siguiente sustitución de la línea 21 es necesaria:

```
frontend.send([args[0],'','NoWorkers'])
```

- Esto asegura que ningún cliente estará en la cola "clients" y que recibirán la respuesta "NoWorkers" cuando sea necesario.

Así, la nueva versión de este broker es:

```
01:  // ROUTER-ROUTER request-reply broker in NodeJS
02:  var zmq      = require('zmq')
03:    , frontend = zmq.socket('router')
04:    , backend  = zmq.socket('router');
05:
06:  var fePortNbr = 8059;
07:  var bePortNbr = 8060;
08:  var workers = [];
09:
10:  frontend.bindSync('tcp://*:'+fePortNbr);
11:  backend.bindSync('tcp://*:'+bePortNbr);
12:
13:  frontend.on('message', function() {
14:    var args = Array.apply(null, arguments);
15:    if (workers.length > 0) {
16:      var myWorker = workers.shift();
17:      var m = [myWorker,''].concat(args);
```

```
18:       backend.send(m);
19:     } else
20:       frontend.send([args[0],'','NoWorkers']);
21:   });
22:
23:   backend.on('message', function() {
24:     var args = Array.apply(null, arguments);
25:     if (args.length == 3) {
26:         workers.push(args[0]);
27:     } else {
28:         var workerID = args[0];
29:         args = args.slice(2);
30:         frontend.send(args);
31:         workers.push(workerID);
32:     }
33:   });
```

3. Analicemos lo que cada proceso está haciendo en cada instante en la ejecución de todos los componentes:

Time	Worker	Broker	Client 1	Client 2	Client 3	Client 4	Client 5	Client 6	Client 7	Client 8
0	Cli1 req rcvd.	Cli1 req. to worker	Req. sent	Req. sent / 'NoWrk' rcvd.	Req. sent / 'NoWrk' rcvd.	Req. sent / 'NoWrk' rcvd.	Req. sent / 'NoWrk' rcvd.	Req. sent / 'NoWrk' rcvd.	Req. sent / 'NoWrk' rcvd.	Req. sent / 'NoWrk' rcvd.
1	Cli1 replied / Cli2 rcvd.	Cli2 req. to worker	Reply rcvd.	Req. resent	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.
2	Cli2 replied / Cli3 rcvd.	Cli3 req. to worker		Reply rcvd.	Req. resent	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.
3	Cli3 replied / Cli4 rcvd.	Cli4 req. to worker			Reply rcvd.	Req. resent	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.
4	Cli4 replied / Cli5 rcvd.	Cli5 req. to worker				Reply rcvd.	Req. resent	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.
5	Cli5 replied / Cli6 rcvd.	Cli6 req. to worker					Reply rcvd.	Req. resent	Req. resent / 'NoWrk' rcvd.	Req. resent / 'NoWrk' rcvd.
6	Cli6 replied	Cli6 reply forw.						Reply rcvd.	Error shown to user	Error shown to user

Como se puede observar en la tabla, cuando la ejecución empieza, uno de los ocho clientes consigue entregar primero su petición al broker. Llamemos a este cliente "Client 1"- El resto de peticiones reciben un "NoWorkers" como respuesta. Esto provoca que todos estos clientes reprogramen el envío de su petición. Consideremos que en este mismo instante se produce la respuesta del worker a la petición de "Client 1". Este recibe la respuesta y los otros siete están enviando su petición. Uno de ellos encontrará al worker en estado disponible. Los seis restantes obtendrán la respuesta "NoWorkers". Han usado el primero de sus cinco reintentos.

En el instante 2, el cliente 2 recibe su respuesta y los 6 restantes envían de nuevo su petición. Uno de ellos será servido. Esto significa que en cada instante se sirve un cliente. Dado que cada cliente envía su petición original más cinco reintentos, ninguno muestra error alguno hasta el instante 6. En ese instante 6 clientes han obtenido su respuesta y sólo se mostrarán dos errores.

PARTE 2 (2 PUNTOS: 0.5 PUNTOS POR CUESTIÓN)

Supongamos un sistema en el que se utilicen las versiones originales de los programas mostrados en la Parte 1 (es decir, *myclient.js, mybroker.js* y *myworker.js*). Explica qué sucede cuando los procesos que se mencionan en las cuestiones siguientes son iniciados y sus respectivas ejecuciones terminen o permanezcan suspendidas (debido a que no hay nuevos eventos que procesar).
Para ello, indica si cada proceso ha…:

- **Clientes**: (**a**) sido incapaz de comunicarse con otros procesos, (**b**) dejado su mensaje en una cola del broker, (**c**) obtenido una respuesta, o (**d**) terminado su ejecución de manera abrupta.
- **Trabajadores**: (**a**) sido incapaz de comunicarse con otros procesos, (b) dejado su identidad en una cola del broker, (**c**) transmitido satisfactoriamente su respuesta al cliente adecuado, o (**d**) terminado su ejecución de manera abrupta.
- **Brokers**: (**a**) sido incapaz de gestionar los mensajes enviados por los clientes, (**b**) sido incapaz de gestionar los mensajes enviados por los trabajadores, (**c**) funcionado sin ningún problema hasta el momento, o (**d**) terminado su ejecución de manera abrupta.

Los casos a considerar son (cada uno asume que no hay procesos en marcha):

1. **Se inician 3 clientes. Un segundo después, se inicia el bróker. No se lanza ningún trabajador.**

   Clientes (b): Todos los clientes han insertado en la cola "clients" su petición. Están esperando su respuesta que llegará si en un futuro próximo se ponen workers en funcionamiento. La conexión con el broker tardará cierto tiempo, pero finalmente se producirá sin error ni excepción.
   Broker (c): Ha recibido tres peticiones de los clientes (una por cada cliente) y las ha almacenado en el vector "clients". Dado que no hay ningún worker esas peticiones permanecen en el vector. El broker funciona correctamente y enviará esas peticiones cuando se ponga un worker en ejecución.

2. **Se inician 2 trabajadores. Tras 3 segundos, se inician 3 clientes. No empieza ningún bróker.**

   Clientes (a): Los clientes han intentado conectarse al broker y han enviado sus peticiones. Esas peticiones serán procesadas cuando se inicie un broker. Por el momento, ningún cliente ha podido comunicar con ningún componente.
   Trabajadores (a): Los workers han intentado conectar con el broker y han enviado sus mensajes de registro inicial. Estos intentos serán procesados cuando se ponga en ejecución un broker. Por el momento, ningún worker se ha podido comunicar con ningún componente.

3. **Se inicia un bróker A. Un segundo después, se inicia un bróker B. De momento no se lanza ningún cliente ni trabajador.**

   Broker A (c): El primer broker se ha iniciado y ejecuta la instrucción BIND correctamente en sus dos sockets ZMQ de tipo ROUTER. Se encuentra esperando mensajes por esos sockets. Su comportamiento es correcto hasta el momento.
   Broker B (d): El segundo broker generará una excepción cuando intente ejecutar la instrucción bind sobre el puerto previamente usado por el anterior broker. Como esta excepción no es tratada en ninguna parte del código, este componente terminará su ejecución.

4. **Se inicia un bróker. Un segundo después, 2 trabajadores A y B son iniciados, en ese orden. 2 segundos más tarde, se lanza un cliente.**

   Broker (c): Funciona correctamente.
   Trabajadores (b): Ambos se han registrado en el broker. El trabajador A ha respondido a la petición del cliente. Después, su ID se almacena en el vector "workers". El ID del trabajador B pasa al vector "workers" y no recibe ninguna petición de cliente alguno. Por tanto, el ID de ambos workers permanece en el vector "workers".
   Cliente (c): El cliente recibe una respuesta a su petición por parte del trabajador A.

PARTE 3 (2 PUNTOS: 0.5 PUNTOS POR ACIERTO, -0.167 PUNTOS POR RESPUESTA ERRÓNEA)

Selecciona la opción correcta. Hay una sola opción correcta en cada cuestión.

1. Sobre la propuesta de solución para latido, indíquese cuál de las siguientes afirmaciones es CIERTA:

	El bróker envía un mensaje a todos los workers, que deben contestar dentro de un plazo de tiempo. El broker no hace un broadcast de ningún mensaje a todos los workers. Tan solo reenvía cada petición de entrada a un worker.
X	Los workers con trabajo deben contestar con el resultado dentro de un plazo de tiempo. El broker establece un 'timeout' para cada petición reenviada. Si la respuesta correspondiente no se recibe en el intervalo marcado, el broker considera que el worker ha fallado y reenvía esa petición a cualquiera de los brokers disponibles. Si no hay workers disponibles, este reenvío se sitúa en el vector "clients".
	Los clientes reenvían las peticiones no finalizadas dentro del plazo de tiempo. Los clientes no reenvían, es el broker el que controla un plazo de tiempo de respuesta por parte de los workers.
	Los workers envían periódicamente mensajes al bróker para confirmar que se encuentran en funcionamiento. El worker no envía mensajes periódicos.

2. Sobre la propuesta de solución para clases de trabajo, indíquese cuál de las siguientes afirmaciones es FALSA:

	Salvo el bróker, todos los demás componentes se limitan a un único tipo de trabajo. Cierto. En la versión original, tanto clientes como workers, una vez iniciados sólo son capaces de gestionar un único tipo de trabajo.
	En el primer mensaje enviado por un worker al bróker, el primero informa acerca del tipo de trabajo que puede atender. Cierto. Los trabajadores se identifican con su tipo de trabajo al broker en el primer mensaje que envían.
X	En el primer mensaje enviado por el bróker a un worker, el primero le informa acerca del tipo de trabajo que debe atender. Falso. El broker sólo reenvía la petición del cliente a un worker.
	Modificando únicamente el código del cliente, éste podría cambiar el tipo de trabajo para cada mensaje sin provocar que el sistema falle. Cierto. Esta modificación es posible. Dado que el cliente incluye el tipo de trabajo en un segmento de su mensaje (en el último de cada petición), si se extiende el código del cliente se puede especificar un tipo de trabajo diferente para cada petición.

3. El bróker gestiona dos colas relacionadas con clientes y trabajadores (workers). Selecciona la afirmación FALSA:

X	La cola de clientes puede disponer de dos elementos que únicamente se diferencien en el contenido del mensaje. Falso. Esto implicaría que hay dos peticiones del mismo cliente en la cola "clients". Esto no puede suceder, dado que un cliente sólo envía una petición cada vez.

	Cuando una cola tiene elementos, la otra se encuentra vacía. Cierto. La cola "clients" sólo se usa cuando no hay ningún worker disponible (es decir que no hay workers en la cola "workers"). Análogamente sólo se encolan workers en la cola worker cuando no hay peticiones pendientes de servir (la cola clients está vacía).
	No todos las peticiones de clientes pasan por su cola clients[]. Cierto. Las peticiones de los clientes se mantienen en la cola "clients" siempre que no haya workers disponibles. Si al recibir la petición del cliente hay algún worker disponible, esa petición no pasa por la cola "clients".
	No todos los mensajes de petición (u ofrecimiento inicial) de servicio de trabajadores pasan por su cola workers[]. Cierto. Si existe una petición de algún cliente, previa al mensaje enviado por el worker, entonces al worker se le asigna esa petición sin insertar su ID en la cola "workers".

4. Indica si en nuestro sistema client-broker-worker el worker puede enviar mensajes con diferente cantidad de segmentos:

	No, porque el código del cliente no está preparado para este caso. Falso. El código del cliente puede gestionar las respuestas cuando contienen diferentes cantidades de segmentos. Los clientes sólo están interesados en un segmento del mensaje. Sin embargo, no producen ninguna excepción cuando el mensaje está vacío o tiene más segmentos.
	No, porque el código del bróker no está preparado para este caso. Falso. El broker está preparado para procesar mensajes de diferentes tipos del worker: su mensaje inicial de disponibilidad y las respuestas a los clientes. Ambos tienen un número distinto de segmentos.
X	Sí, porque a veces el mensaje del worker no contiene la respuesta a una solicitud. Cierto. El mensaje inicial del worker tiene tres segmentos, mientras que las respuestas a los clientes tienen más de tres segmentos.
	Sí, porque depende del número de segmentos que tenga la petición que llegue al worker. Falso. Los workers sólo procesan uno de los segmentos en la petición de los clientes. Por tanto la respuesta de los workers no depende de la cantidad de segmentos de la petición del cliente.