TSR: Actividades del Tema 3 con ZMQ

Agrupadas en tres bloques, de manera que el primero recoge ejercicios más sencillos mientras que los otros dos exigen soluciones más elaboradas.

Primer Bloque (1 a 4)

ACTIVIDAD 1

OBJETIVO: Experimentar con el uso de bind() y connect()

ENUNCIADO: Tomando como base el ejemplo de la transparencia 48:

```
// Client
var zmq = require('zmq')
var rq = zmq.socket('req')

rq.connect('tcp://127.0.0.1:8888')

rq.send('Hello')

rq.on('message', function(msg) {
   console.log('Response: ' + msg)
})
```

```
// Server
var zmq = require('zmq')
var rp = zmq.socket('rep')

rp.bind('tcp://127.0.0.1:8888',function(err) {
   if (err) throw err
})

rp.on('message', function(msg) {
   console.log('Request: ' + msg)
   rp.send('World')
})
```

1. Modificar el cliente para que envíe un mensaje periódicamente.
2. Realizar una nueva modificación del cliente. Ahora el mensaje debe incorporar un número de secuencia, para que así se pueda distinguir entre cada mensaje recibido.
3. Ejecutar el cliente modificado y el servidor en procesos diferentes.
    a. Modifique el servidor de manera que termine (sin responder) tras haber recibido la décima petición.
    b. Reinicie el servidor.
    c. Describa qué ocurre en el cliente.

Modifiquemos ahora ambos programas como sigue:

- El cliente hará un bind() de su socket, en lugar de un connect().
- El servidor realizará un connect(), en lugar de un bind().

Ahora, al igual que antes, ejecute cliente y servidor. Termine el servidor. Lance el servidor de nuevo.

4. Describa las diferencias en el comportamiento del cliente, en caso de que observe alguna.
5. Discuta qué ventajas e inconvenientes aporta estos cambios en los bind() y connect().
6. Presente el código resultante.

ACTIVIDAD 2

OBJETIVO: Adaptar el patrón petición-respuesta de los sockets al uso de promesas.

ENUNCIADO: En el patrón petición-respuesta, el agente que usa el socket **req** es el cliente, mientras que el agente con el socket **rep** es el servidor.

Como ya se ha presentado en clase, el API gestiona el mensaje de respuesta como un evento ("message") y los problemas potenciales con otro evento ("error").

Se solicita la escritura de un módulo que tome como base **zmq** (que podría llamarse **pzmq**). Este módulo debe proporcionar un objeto similar a los del módulo **zmq**. Sin embargo, cuando se use el método **socket** sobre ese objeto, se deberá evaluar si el socket es de tipo **req**. Si lo es, el socket retornado debe admitir un método **request**() con una signatura que admita un número variable de segmentos en el mensaje. Así:

```
var reply = sock.request(segment1, ..., segmentn)
```

Donde **reply** es una promesa que será resuelta cuando el mensaje de respuesta llegue al socket **req** o cuando un evento "error" sea generado.

En el primer caso, el valor de la promesa resuelta es el mensaje de respuesta (como un vector de segmentos). En el segundo caso la promesa será rechazada con el valor del evento "error" generado por el socket **req** subyacente.

ACTIVIDAD 3

OBJETIVO: Recuperarse de fallos potenciales en los socket **rep**.

ENUNCIADO: En la actividad 1 se observó que el fallo de un servidor antes de contestar una petición puede bloquear a un cliente **req**.

En la actividad 2 se ha visto cómo convertir el patrón de intercambio de mensajes asíncrono en un patrón basado en promesas.

Aprovecharemos lo aprendido en la actividad 2 para manejar los fallos que bloqueen al proceso cliente. Con este fin, se solicita el desarrollo de una extensión para el módulo **pzmq**, donde el método **request()** introducido en la actividad 2 reciba un parámetro adicional: un timeout (especificado en segundos). Ahora la signatura de ese método será:

```
var reply = sock.request(tmout, segment1, ..., segmentn)
```

Su comportamiento debe ser:

1. Si `tmout==0` , entonces realizará la misma función que en la actividad 2.
2. En otro caso, cuando transcurran `tmout` segundos sin que haya ningún error ni se haya recibido ningún mensaje, el método cerrará el socket y lo abrirá de nuevo de la misma manera, utilizando los mismos argumentos que en la llamada original. Adicionalmente, la promesa `reply` será rechazada con un valor "TMOUT".

Repita el experimento de la actividad 1 utilizando la nueva API. Describa si el nuevo comportamiento es diferente al original. En ese caso, describa qué ha mejorado o empeorado.

ACTIVIDAD 4

OBJETIVO: Aplicar la técnica de la actividad 2 a los sockets **dealer**. Organizar de manera más sencilla las peticiones concurrentes mediante sockets **dealer**.

ENUNCIADO: Los sockets **dealer** pueden ser utilizados allí donde los sockets **req** tengan sentido. Sin embargo, los sockets **req** utilizan un segmento delimitador en el mensaje (tal como esperan los sockets **rep**), mientras que los sockets **dealer** no realizan esta gestión.

En esta actividad se solicita una extensión de la funcionalidad del módulo **pzmq** para añadir un nuevo tipo de socket al que llamaremos **dealerReq**.

Los sockets **dealerReq** se comportan como una mezcla entre **dealer** y **req**, pero su funcionalidad se implantará mediante un método **request()** similar al de la actividad 2, basado en los socket **dealer** de 0MQ.

Observe que los sockets **dealer** permiten el envío concurrente de mensajes de petición (cosa que no ocurre en los sockets **req**). Por tanto, su implementación debe soportar eso.

PISTA: Tendremos sólo un socket **dealer** subyacente, que podrá enviar muchas peticiones concurrentes, incluso sobre sockets **rep** diferentes. ¿Cómo se podrá distinguir a qué petición corresponde cada respuesta?

1. Observe que con sockets **req** esto era sencillo pues en cada momento solo habrá una petición pendiente. A esa petición estará ligada la respuesta actual. Con un socket **dealer** esto no se respetará siempre.
2. Observe que ese problema debe ser gestionado adecuadamente, ya que el API facilitada "promete" la respuesta para una petición determinada… (no sólo un evento "message" genérico del socket 0MQ).

Segundo Bloque (5 a 8)

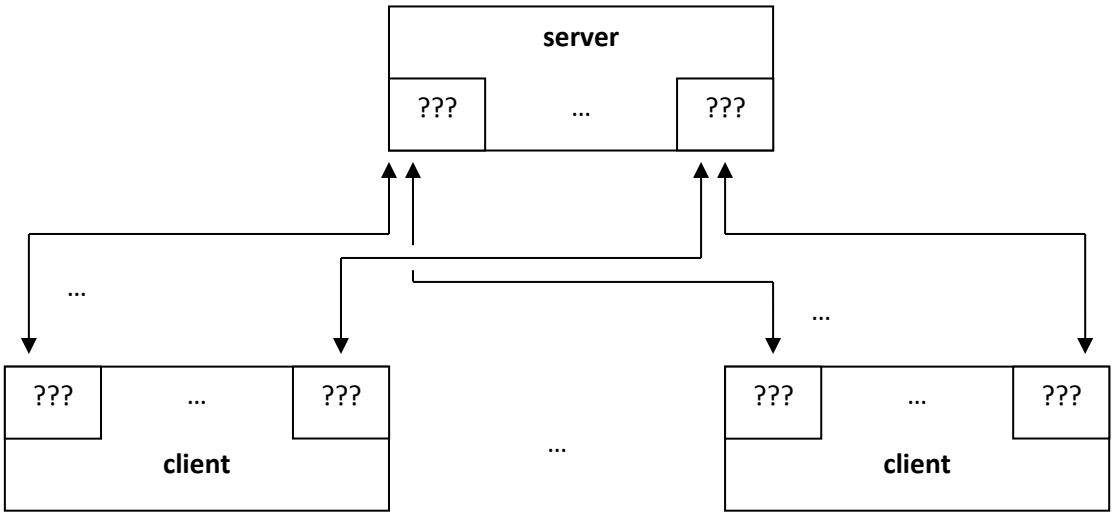
En la primera actividad de este bloque, se plantea el diseño e implementación de una aplicación de **chat** muy sencilla: su servidor atenderá a un número variable de clientes que participarán en una única conversación. En la segunda actividad, se pide diseñar e implementar una aplicación de **descarga de ficheros**, la cual también será muy simple: se atenderán peticiones de descarga de ficheros de texto, exclusivamente. Las actividades restantes (7 y 8) plantean un **servicio de cómputo** de funciones matemáticas, basado en broker, usando 0MQ para la comunicación entre servidores y clientes del servicio.

ACTIVIDAD 5

Se desea diseñar un **servicio de chat** usando 0MQ en el envío de los mensajes entre los usuarios del servicio. Para ello, hay que diseñar dos módulos: un módulo **servidor**, que distribuya los mensajes entre los usuarios; y un módulo **cliente**, que ejecutará cada uno de los usuarios que se conecte al servidor.

El funcionamiento de los módulos debe ajustarse a lo siguiente. Una vez conectado al servidor, el módulo cliente ha de poder enviar, en todo momento, mensajes al servidor, con la garantía de que los mismos serán recibidos por todos los demás clientes que estén conectados. Por su parte, el servidor tiene que aceptar cualquier mensaje que le envíe cualquier cliente conectado, y retransmitirlo a todos los clientes. El servidor también ha de notificar la incorporación de un nuevo cliente a los clientes ya conectados, y la desconexión de un cliente a los otros clientes que permanezcan.

En primer lugar, hay que elegir el número y tipo de **sockets 0MQ** que sean más adecuados para satisfacer las comunicaciones requeridas en el servicio de chat. En la siguiente figura, los interrogantes representarían sockets cuyo tipo se debe decidir.



Una vez definido el diseño de los módulos, se puede empezar con su implementación.

En primer lugar, en el módulo servidor, será necesario declarar variables para los sockets elegidos y realizar su *binding*, para que estén disponibles para los clientes. En el siguiente fragmento de código, se muestra la estructura que tendría esta parte del servidor (se indican puntos suspensivos para señalar código a completar).

```
...
// *** chat server variables

var zmq = require('zmq')
  , ... = zmq.socket( ... )
  , ... = zmq.socket( ... )
  , ...

// *** binding sockets

... .bind('tcp://*:' + ... , function(err) {
  ...
})

... .bind('tcp://*:' + ... , function(err) {
  ...
})

...
```

Análogamente, en el módulo cliente, será necesario declarar variables para los sockets elegidos y realizar su *connect*, para comunicarse con el servidor y, a través de éste, con los demás clientes. En el siguiente fragmento de código, se muestra la estructura que tendría esta parte del cliente (se indican puntos suspensivos para señalar código a completar).

```
...
// *** chat client variables

var zmq      = require('zmq')
  , ...      = zmq.socket( ... )
  , ...      = zmq.socket( ... )
  , ...

// *** connecting sockets

... .connect( ... )
... .connect( ... )

...
```

A continuación, el cliente debería enviar un mensaje de solicitud de conexión al servidor (en nuestro caso de estudio, esta solicitud será aceptada siempre). En dicho mensaje, el cliente proporcionaría su identificador, para que sus intervenciones en el chat sean reconocidas por los otros clientes. Esta identidad podría facilitarse como un argumento en la invocación del módulo cliente, o bien ser generada aleatoriamente dentro del propio módulo. En el siguiente fragmento de código, se muestra la estructura que tendría esta parte del cliente (de nuevo, se indican puntos suspensivos para señalar código a completar).

```
...

// *** setting client identity
// *** supposing a randNumber function that generates random
numbers

if ( ... )  ... .identity = process.argv[ ... ]
else        ... .identity = 'client-' + randNumber( ... )


// *** sending connect message

var connect_msg = {'type': ... , 'identity': ... }
... .send( JSON.stringify(connect_msg) )

...
```

A partir de ese momento, el cliente debería poder enviar los mensajes que quisiera, así como recibir todos los mensajes que otros clientes enviaran. Para lo segundo (recibir mensajes de los demás), debería conectarse a la escucha de algún socket del servidor, y mostrar en su ventana terminal los mensajes recibidos.

```
// *** receive everything

...

... .on('message', ...
  console.log( ... )
...
```

Para lo primero (escribir y enviar sus propias intervenciones en el chat), el cliente debería tener habilitada la entrada estándar, construir un mensaje de chat que contuviera, al menos, su identidad y el texto de su intervención, y enviar este mensaje al servidor usando el socket que corresponda. A modo de orientación:

```
// *** sending chat messages

process.stdin.resume()
process.stdin.setEncoding('utf8')

process.stdin.on('data', ...
  var chat_msg = { ... }
  ... .send( ... )
...
```

Finalmente, el cliente debería poder notificar al servidor cuando se desconecta de la conversación. Para ello, se debe implementar que, cuando el cliente cierre la entrada estándar (pulsando Ctrl+D), se envíe un mensaje especial de desconexión al servidor. Además, al cerrar la aplicación cliente (pulsando Ctrl+C), se deberían cerrar los sockets que el cliente hubiera usado.

En lo que concierne al servidor, su tarea consiste en estar a la espera de mensajes y retransmitirlos adecuadamente. El servidor usará algún socket para mantenerse a la escucha de modo indefinido y, cada vez que reciba algo, lo identificará, construirá el mensaje que

corresponda, y lo enviará a todos los clientes conectados. El servidor debe identificar cualquier mensaje recibido, dado que existirán 3 tipos de mensaje:

- Mensaje de conexión. Es el mensaje inicial que envía cualquier cliente, informando de su identidad. Si, por ejemplo, el cliente se identificara como "*Juan Nadie*", el servidor debería enviar a todos los clientes un mensaje con un texto como el siguiente: "*Server > chat-client Juan Nadie connected!*".

- Mensaje de chat. Es el mensaje que contiene una intervención en la conversación. El servidor ha de reenviarlo a todos los clientes, indicando la identidad del emisor. Si, por ejemplo, el cliente "*Juan Nadie*" escribe, en su mensaje de chat, "*No tengo nada que decir*", el servidor enviará a todos un mensaje con un texto como el siguiente: "*Juan Nadie > No tengo nada que decir*".

- Mensaje de desconexión. Es el mensaje final que puede enviar cualquier cliente, si desea avisar de que abandona la conversación. El servidor debería enviar a todos un mensaje como el siguiente: "*Server > chat-client Juan Nadie disconnected!*".

Así pues, a modo de plantilla de esta parte del servidor, considérense las siguientes líneas:

```
// *** listening on input socket
// *** identifying message type
// *** broadcasting on output socket

... .on('message', function(msg) {
  var ...  = JSON.parse(msg)
    , type = ...
    , ...  , ...  , ...
  if ( type == ... )
    ...
  if ( type == ... )
    ...
  if ( type == ... )
    ...
  ... .send( ... )
})
```

Finalmente, al cerrar la aplicación servidor (pulsando Ctrl+C), se deberían cerrar los sockets que el servidor hubiera usado.

ACTIVIDAD 6

Se desea diseñar un **servicio de descarga de ficheros de texto** usando 0MQ en el envío de los mensajes entre el servidor de ficheros y los usuarios del servicio. Para ello, hay que diseñar dos módulos: un módulo **servidor**, que reciba las peticiones de descarga que envíen los usuarios conectados, y las atienda, enviando los ficheros solicitados; y un módulo **cliente**, que ejecutará cada uno de los usuarios que se conecte al servidor.

El funcionamiento de los módulos debe ajustarse a lo siguiente:

- El servidor atenderá cada petición. Si el fichero solicitado no existe, contestará con algún mensaje especial (por ejemplo, un mensaje vacío). Si el fichero existe, lo leerá. El fichero leído no será enviado como contenido de un único mensaje. El servidor dividirá el contenido en bloques de 1K, y enviara tantos mensajes como sean precisos, en función del tamaño del fichero (todos los mensajes contendrán 1K de datos, excepto el último que contendrá el resto).

- Una vez conectado al servidor, el módulo cliente podrá escribir en la entrada estándar el nombre del fichero que desea descargar, y quedará a la espera de que la descarga se complete (o se informe de algún error: por ejemplo, inexistencia del fichero solicitado), tras lo cual debería poder repetir la operativa de indicar nombre de fichero y esperar su descarga tantas veces como quiera. Dado el modo de envío del fichero por parte del servidor, el módulo cliente deberá ir escribiendo los bloques de 1K (transmitidos en cada mensaje) en el fichero local asociado a la descarga en curso, y procederá a cerrar el fichero cuando reciba el último mensaje asociado al mismo.

Además, en el diseño e implementación del servicio, se tendrán en cuenta las siguientes consideraciones:

- Para simplificar la gestión del servicio, se supondrá que cualquier cliente ya conoce los nombres de los ficheros disponibles en el servidor.

- Los clientes solicitarán ficheros usando nombres de ruta relativos al directorio donde esté el servidor. Así, si un cliente solicita el fichero "*kipling.txt*", dicho fichero debería existir en el directorio donde esté el módulo servidor; y si solicitara el fichero "*textos/kipling.txt*", el servidor buscaría el fichero "*kipling.txt*" en el subdirectorio "*textos*".

- El único tipo de error que comprobará el servidor es la petición de descarga relativa a un fichero inexistente (respondiendo al cliente, como se ha indicado antes, con algún mensaje especial).

- En este sencillo servicio de descarga, se supondrá que no se presentarán otros posibles errores (como denegación de acceso al fichero solicitado, solicitud de un fichero que no sea de texto, formato incorrecto del mensaje de petición del cliente, etc.). Por tanto, el módulo servidor no tendrá que hacer ningún tratamiento de los mismos. En una aplicación más completa y funcional, estas excepciones deberían tratarse.

ACTIVIDAD 7

Se desea diseñar un **servicio de cómputo de funciones matemáticas** usando 0MQ en el envío de los mensajes entre los servidores de cómputo y los usuarios o clientes del servicio, mediante una arquitectura centralizada en un broker. Para ello, hay que diseñar tres módulos:

- Un módulo **broker** que, por una parte, reciba las peticiones de los clientes y reenvíe dichas peticiones, de modo equitativo, a los servidores de cómputo, y, por otra parte, reciba los resultados de los cálculos y los reenvíe a los clientes que los solicitaron. El funcionamiento del servicio sólo requerirá la ejecución de un módulo broker.

- Un módulo **servidor** que, a través del broker, reciba las peticiones de cómputo de los clientes, realice los cálculos solicitados y envíe, también a través del broker, los resultados a los clientes. El servicio debe diseñarse de manera que pueda funcionar con cualquier número de servidores.

- Un módulo **cliente**, que ejecutará cada uno de los usuarios que se conecte al broker. Este módulo enviará las peticiones de cálculo de un usuario y quedará a la espera de recibir los mensajes correspondientes a los resultados. El servicio debe diseñarse de manera que pueda funcionar con cualquier número de usuarios conectados.

Para simplificar el problema, se considera que el cómputo de las funciones matemáticas se limita a las incluidas en el siguiente módulo, **myMath.js**. Este módulo deberá usarse en la resolución del ejercicio, y no deberá modificarse.

```
1.  function fibo(n) {
2.     return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
3.  }
4.
5.  function fact(n) {
6.     return (n<2) ? 1 : n * fact(n-1)
7.  }
8.
9.  function prim(n) {
10.    for (var i=2; i<=Math.sqrt(n); i++)
11.       if (n%i === 0) return false
12.    return true
13. }
14.
15. function calculate(f, n) {
16.   var res = 0
17.   switch (f) {
18.     case 'fibo': res = fibo(n); break
19.     case 'fact': res = fact(n); break
20.     case 'prim': res = prim(n); break
21.     case 'sen':  res = Math.sin(n); break
22.     case 'cos':  res = Math.cos(n); break
23.     case 'tan':  res = Math.tan(n); break
24.     case 'log':  res = Math.log(n); break
25.     case 'exp':  res = Math.exp(n); break
26.     case 'raiz': res = Math.sqrt(n); break
27.     default:     res = NaN
28.   }
29.   return res
30. }
31.
```

```
32. ids = ['fibo', 'fact', 'prim', 'sen', 'cos', 'tan',
33.        'log', 'exp', 'raiz', 'strange']
34.
35. exports.eval = calculate
36. exports.funcs = ids
```

El módulo **myMath** exporta una función y un array. Los módulos cliente y servidor tendrán que importar el módulo. En concreto, la función **eval** será usada por el módulo servidor para calcular las funciones solicitadas por los clientes, y el array **funcs** será usado por el módulo cliente para especificar la función que se pide calcular.

Además, el funcionamiento de los módulos debe ajustarse a lo siguiente:

- El **broker** dispondrá de dos puertos. Un puerto servirá para comunicarse con los usuarios del servicio (módulos cliente), el otro puerto para comunicarse con los servidores de cómputo. Cualquier petición de un cliente será redirigida hacia alguno de los servidores conectados, siguiendo un reparto equitativo de la carga (estrategia round-robin). Cualquier respuesta de un servidor será reenviada al cliente que la solicitó. El broker será un módulo que se ejecute indefinidamente.

- El **servidor** dispondrá de un puerto a través del cual el broker le hará llegar peticiones de cómputo y el mismo enviará las respuestas correspondientes. El servidor realizará el cálculo indicado en la petición recibida y construirá un mensaje de respuesta en el que incluirá, al menos, el resultado del cálculo. Dado que el cálculo de cualquier función del módulo **myMath** es inmediato (con excepción de la función **fibo** para números mayores a 40), se sugiere usar una temporización (con la función **setTimeout**) en el envío de los mensajes de respuesta del servidor. Así, la simulación planteada en este ejercicio será algo más realista. El servidor será un módulo que se ejecute indefinidamente.

- El **cliente** dispondrá de un puerto para comunicarse con el broker, lo usará para enviar sus peticiones y para esperar (escuchar) las respuestas. El mensaje de petición deberá incluir, al menos, un identificador de función a evaluar y un valor (el argumento para la función). Los identificadores de función serán, obligatoriamente, valores del array **funcs**, importado del módulo **myMath**[1]. Se sugiere implementar el cliente de manera que lance automáticamente un cierto número de peticiones[2], espere a la recepción de las respuestas a todas esas peticiones, mostrando los resultados en la salida estándar. Si se sigue esta sugerencia, el cliente terminará su ejecución cuando reciba y muestre todas las respuestas.

[1] La última componente del array **funcs**, el identificador **strange**, pretende representar a una función que no puede evaluar el servidor de cómputo. Adviértase que para cualquier valor **n**, el resultado de **eval(strange, n)** es **NaN**, Not a Number.

[2] Se pueden elegir aleatoriamente tanto las funciones a evaluar como los argumentos de estas funciones, si bien es recomendable no invocar la función **fibo** con un valor mayor a 40, dado el coste exponencial que tiene la implementación recursiva de la función.

ACTIVIDAD 8

Esta actividad consiste en una ampliación o mejora del **servicio de cómputo de funciones matemáticas** desarrollado en la actividad anterior. Se mantiene la arquitectura centralizada en un broker que permite interconectar un número variable tanto de módulos cliente como de módulos servidores o trabajadores.

La diferencia reside en el hecho de que habrá dos tipos de servidores. Existirán servidores limitados en sus prestaciones, que no serán adecuados para evaluar las funciones costosas (en nuestro caso, las funciones **fibo**, **fact** y **prim** del módulo **myMath**). Y existirán servidores potentes, capaces de procesar en un tiempo razonable cualquier función. Para que el broker pueda saber el tipo de cada servidor conectado, éste deberá enviar un mensaje especial de conexión al broker en el que incluya su identificador y algún valor de tipo *flag* que especifique si es un servidor limitado o potente.

En los mensajes de peticiones de los clientes, además de incluir el identificador de función y su argumento, se podría incluir también un valor *flag* que especificara si la función es costosa (es decir, si es **fibo**, **fact** o **prim**) o si no lo es (cualquiera de las otras, importadas del módulo **Math**). Para poder redirigir adecuadamente las peticiones hacia los servidores, el broker debería disponer de alguna estructura de datos interna donde mantuviera las identidades y tipos de los servidores conectados.

Si la función a evaluar en una petición es costosa, el broker enviaría esa petición hacia alguno de los servidores potentes conectados. Si la función a evaluar en una petición no es costosa, el broker podría enviar la petición hacia cualquiera de los servidores conectados. Hecha esta distinción, el broker debería procurar mantener el reparto equitativo de la carga. Así pues, la evaluación de las funciones “ligeras en cómputo” debería repartirse equilibradamente entre todos los servidores, y la evaluación de las funciones “pesadas en cómputo” debería repartirse equilibradamente entre los servidores potentes.

Las modificaciones planteadas en esta actividad, respecto a la anterior, podrían implicar cambios en los patrones de comunicación (tipos de sockets 0MQ a usar) y en la estructura de los mensajes a construir y transmitir (mensajes multi-parte). Al igual que en las otras actividades de este boletín, antes de ponerse a implementar nada, es muy recomendable analizar el problema planteado y tomar las adecuadas decisiones de diseño.

Tercer Bloque (9 a 12)

ACTIVIDAD 9

OBJETIVO: Gestionar los patrones de comunicación DEALER-REP y DEALER-DEALER de manera adecuada.

ENUNCIADO: Considere los siguientes programas NodeJS:

```
1  // FILE: client.js
2  const zmq = require('zmq')
3  const rq = zmq.socket('req')
4  rq.connect('tcp://127.0.0.1:8888')
5  rq.connect('tcp://127.0.0.1:8889')
6
7  let i=0
8  function mySend() {
9     rq.send(++i); console.log(‘Request: ‘+i)
10    rq.send(++i); console.log(‘Request: ‘+i)
11 }
12 rq.on('message', function(req, res) {
13    console.log('Request: ‘+req+’ Response: ' + res)
14 })
15 setInterval( mySend, 100 )
```

```
1  // FILE: server.js
2  const zmq = require('zmq')
3  const path = require('path')
4  const rp = zmq.socket('rep')
5
6  // Check whether we received the correct number of arguments.
7  if (process.argv.length < 3) {
8     // If not, print a usage message and exit.
9     console.log('Usage: node %s port-number', path.basename(process.argv[1]))
10    process.exit(1)
11 }
12
13 // Get the port number from the command-line arguments.
14 let port = process.argv[2]
15
16 rp.bind('tcp://127.0.0.1:'+port,
17 function(err) {
18    if (err) throw err
19 })
20
21 rp.on('message', function(msg) {
22    console.log('Request: ' + msg)
23    rp.send([msg,parseInt(msg)*2])
24 })
```

El proceso cliente envía mensajes que contienen un número entero. El servidor responde a cada mensaje devolviendo ese número multiplicado por 2. El programa servidor recibe el número de puerto a usar desde la línea de órdenes.

Así, para iniciar un ejemplo de ejecución, deberíamos utilizar algo similar a:

- Para iniciar el cliente: `node client`
- Para iniciar los servidores: `node server 8888 & node server 8889 &`

Responda las siguientes cuestiones:

1. Escriba una variación de client.js llamada clientD.js que utilice un socket DEALER en vez de un socket REQ. Debe ser capaz de interactuar sin problemas con dos servidores server.js que atiendan los puertos 8888 y 8889 (es decir, con los servidores originales).
2. Escriba una variación del programa server.js llamada serverD.js que utilice un socket DEALER en vez de un socket REP. Dos servidores de este tipo deben interactuar sin problemas con el nuevo programa clientD.js.
3. Observe, sin embargo, que los sockets DEALER ofrecen un comportamiento preconfigurado en caso de conectarse con varios procesos. En esa situación un socket DEALER envía sus mensajes siguiendo un orden circular sobre todas las conexiones que haya establecido. Por ello, debería tenerse cuidado a la hora de usar un socket DEALER en un proceso servidor si se pretende que este interactúe con múltiples clientes. Explique qué sucedería en ese escenario y justifique si tendría sentido el comportamiento resultante. Para ello, asuma un escenario con tres procesos clientD y un proceso serverD. ¿Se comportan correctamente? ¿Por qué?
4. Amplíe el programa clientD.js original, recibiendo ahora la longitud del intervalo de envío como un argumento desde la línea de órdenes. Además, en cada console.log() utilizado en el programa clientD.js, incluya como información inicial el PID de ese proceso cliente. Para ello, puede utilizar la propiedad process.pid. Utilice diferentes intervalos de envío en cada uno de los clientes mencionados en el apartado anterior (por ejemplo, 100, 210 y 450 ms, respectivamente) y repita el experimento. ¿Se comportan ahora correctamente? ¿Por qué?

ACTIVIDAD 10

OBJETIVO: Verificar las ventajas de la comunicación asincrónica.

ENUNCIADO: A partir de una solución correcta para los dos primeros apartados de la Actividad 9, decida (y justifique con detenimiento) si las siguientes afirmaciones son verdaderas o falsas. Asuma que no había ningún proceso en marcha antes de empezar las pruebas que se citan en cada apartado.

1. Al utilizar esta línea de órdenes, los tres procesos generados no pueden comunicarse:
   `node clientD & node serverD 8888 & node serverD 8889 &`

2. Al utilizar esta línea de órdenes, el proceso cliente es incapaz de interactuar con el servidor:
   `node client & node serverD 8888 &`

3. Al utilizar esta línea de órdenes, ambos procesos se bloquean tras haberse recibido la primera respuesta:
   `node clientD & node server 8888 &`

4. Al utilizar esta línea de órdenes, uno de los tres procesos genera un error (por ejemplo, “dirección ya en uso”) cuando es iniciado:
   `node clientD & node clientD & node server 8888 &`

ACTIVIDAD 11

OBJETIVO: Emular un servicio de pertenencia.

ENUNCIADO: Supongamos un sistema con diez procesos como máximo. Se solicita el desarrollo de un servicio de pertenencia capaz de proporcionar las identidades de todos los procesos de ese sistema que esté en marcha en ese momento.

Para ello, se tendrá que desarrollar un programa que emule a cada uno de esos procesos. Ese programa cumplirá los siguientes requisitos:

- El identificador de proceso (ID) se recibirá como argumento en la línea de órdenes. Debe ser un número natural en el rango 1..10.
- El programa realizará un bind() para uno de sus sockets sobre el puerto 9000+ID. Si se necesitara algún otro socket, se conectará a cada uno de los sockets de los demás procesos.
- Cada proceso debe difundir mensajes de “heart-beat” periódicamente. Por ejemplo, cada segundo. Así, se informa a los demás procesos acerca de la vivacidad del emisor. El contenido del mensaje será el ID del emisor.
- Al recibir un mensaje “heart-beat”, su emisor será considerado como “vivo” por parte del receptor. Además, se establecerá un temporizador (con un intervalo de 1100ms) para esperar el siguiente “heart-beat” de ese mismo emisor. Si alguno de esos temporizadores genera un “timeout”, su proceso asociado se considerará “caído”.
- Cuando el usuario pulse [Ctrl]+[C] (evento ‘SIGINT’ del objeto “process”), el proceso que reciba esa señal mostrará en la pantalla el conjunto actual de procesos “vivos”.
- Todos los procesos puden funcionar sobre el mismo ordenador. En ese caso, utilice la dirección “127.0.0.1” en la URL de los sockets.

Implante ese programa y compruebe su funcionamiento iniciando varios procesos con IDs diferentes. Realice las siguientes pruebas para verificar su correcto funcionamiento:

1. Use [Ctrl]+[C] para obtener el conjunto de pertenencia actual. Inicie otros procesos y compruebe que ese conjunto recoge ahora sus identificadores. Compruebe que cada uno de esos procesos reporta la misma información.
2. Utilizando la orden “top”, averigüe los PIDs de esos procesos. Use la orden “kill” para eliminar algunos de ellos y compruebe que el conjunto de pertenencia reportado por los procesos existentes se ha actualizado adecuadamente. Compruebe de nuevo que todos los participantes reporten el mismo conjunto.
3. Explique qué tipos de sockets han resultado necesarios en su solución y analice si se podría utilizar algún otro tipo de socket para obtener la misma funcionalidad.

ACTIVIDAD 12

OBJETIVO: Emular un servicio de pertenencia.

ENUNCIADO: Se solicita extender la solución a la Actividad 11 de la siguiente manera. Ahora cada proceso del sistema debe ser capaz de actuar también como informador para clientes externos. Estos requisitos describen la extensión a realizar:

- Cada proceso debe proporcionar un “endpoint” donde recibir solicitudes enviadas por los clientes. Se utilizará para ello el puerto número 9100+ID.
- Los mensajes de petición no incluyen ninguna información relevante.
- Los mensajes de respuesta proporcionan la lista de IDs de los procesos actualmente activos.

Habrá que escribir un programa cliente. Este recibe como argumento desde la línea de órdenes el ID del proceso servidor con el que tendrá que interactuar. Una vez iniciado, el cliente enviará un mensaje de petición a ese servidor. En algún momento recibirá su respuesta. Cuando eso ocurra, el cliente mostrará por pantalla la información recibida y finalizará su ejecución.

Compruebe que tanto los procesos clientes como los servidores se comporten correctamente.