TSR: Actividades del Tema 6

ACTIVIDAD 1

Se desea diseñar un **servicio de http** usando el módulo **cluster** de NodeJS. El módulo a implementar deberá cumplir los siguientes requisitos:

1. Lanzar la ejecución de los siguientes procesos: un proceso que sea el cluster master y tantos procesos servidores http como procesadores (cpus) existan en el nodo.

2. Todos los procesos servidores http ejecutarán un sencillo servidor "hola mundo" como el siguiente:

```
1: server = http.createServer(
2:    function (request, response){
3:       response.writeHead(200, {'Content-type': 'text/plain'});
4:       response.end('Hello World!');
5: });
```

   Aviso: Este servidor se proporciona como ejemplo. La resolución de la actividad podría suponer ampliar y/o modificar el código del servidor http a integrar en el cluster.

3. Todos los procesos servidores http escucharán peticiones en un mismo puerto, que se proporcionará como argumento en la línea de comandos.

4. Todos los procesos servidores http notificarán al proceso master, mediante mensajes, todas las peticiones de servicio que hayan atendido.

5. El proceso master almacenará los contadores de las peticiones atendidas por cada servidor http.

6. El proceso master mostrará en consola, periódicamente, los contadores de las peticiones atendidas por cada uno de los servidores http.

El módulo cluster de servidores http que se implemente se podrá probar desde cualquier navegador web. Ahora bien, para poder proporcionar al módulo una cantidad apreciable de peticiones por unidad de tiempo, se deberá probar su funcionamiento usando el programa *http_get_client.js*, cuyo código se muestra a continuación:

```
1: // http_get_client.js
2:
3: if (process.argv.length != 3) {
4:    console.log('usage: node http_get_client port');
5:    process.exit(0);
6: }
7:
8: var http = require('http');
9: var options = {
10:    host: 'localhost',
11:    port: parseInt(process.argv[2]),
12:    path: '/'
13: };
14:
```

```
15: function http_client(){
16:    var req = http.get(options, function(response){
17:       var res_data = '';
18:       response.on('data', function(chunk){
19:          res_data += chunk;
20:       });
21:       response.on('end', function(){
22:          console.log(res_data);
23:       });
24:    });
25:    req.on('error', function(e){
26:      console.log('An error: ' + e.message);
27:    });
28: };
29:
30: for (var i=0; i<10; i++)
31:    setInterval(http_client, 1);
```

El programa *http_get_client.js* enviará 10 peticiones al módulo cluster de servidores http por milisegundo.

ACTIVIDAD 2

Se desea diseñar un **servicio de descarga de ficheros** de texto usando **0MQ** y los módulos **fs** y **cluster** de NodeJS. El módulo a implementar deberá cumplir los siguientes requisitos:

1. Lanzar la ejecución de los siguientes procesos: un proceso que sea el cluster master y tantos procesos servidores de descarga como procesadores (cpus) existan en el nodo.

2. Todos los procesos servidores ejecutarán un servidor como el siguiente (se proporciona un código incompleto, se indican puntos suspensivos donde hay que completar el servidor):

```
 1:    // worker process - create REP socket, connect to DEALER
 2:    var rep = zmq.socket('rep').connect( ... )
 3:
 4:    rep.on('message', function(data) {
 5:       // parse incoming message
 6:       var request = JSON.parse(data)
 7:       console.log( ... )
 8:
 9:       // read file and reply with content
10:       fs.readFile(request.path, function(err, data) {
11:          if (err) {
12:             console.log( ... )
13:             data = ' NOT FOUND'
14:          } else {
15:             console.log( ... )
16:          }
17:          rep.send(JSON.stringify({
18:                   pid: ... ,
19:                   path: ... ,
20:                   data: ... ,
21:                   timestamp: ...
22:          }))
23:       })
24:    })
```

3. El master utilizará sockets **ROUTER** y **DEALER** de 0MQ para gestionar los mensajes de peticiones de descarga (enviados por los clientes) y los mensajes de descarga de ficheros (enviados por los servidores).

4. Se usará el protocolo **TCP** en la conexión entre los sockets REQ de los clientes y el socket ROUTER del cluster. Se usará el protocolo **IPC** en la conexión entre los sockets REP de los servidores y el socket DEALER del master.

5. Cuando el master termine, también deberán cerrarse todos los procesos servidores.

El módulo cluster de servidores de descarga de ficheros deberá atender debidamente las peticiones que le lleguen de clientes que ejecuten el programa ***zmq-requester.js***, cuyo código se muestra a continuación:

```
 1: // zmq-requester.js
 2:
 3: if (process.argv.length != 4) {
 4:    console.log('usage: node zmq-requester port filename');
 5:    process.exit(0);
 6: }
 7:
 8: var zmq      = require('zmq')
 9:   , req      = zmq.socket('req')
10:   , endpoint = 'tcp://localhost:'+process.argv[2]
11:   , filename = process.argv[3]
12:
13: req.connect(endpoint)
14:
15: // send request for content
16: console.log('Sending request for ' + filename)
17: req.send( JSON.stringify({path:filename}) )
18:
19: // handle replies from responder
20: req.on('message', function(data) {
21:    var reply = JSON.parse(data)
22:      , nf    = ' NOT FOUND'
23:      , err   = ( reply.data == nf ) ? nf : ''
24:    console.log('Received reply:',
25:                'file:', reply.path + err,
26:                'from worker:', reply.pid,
27:                'at:', reply.timestamp)
28:    req.close()
29:    process.exit(0)
30: })
```

El programa *zmq-requester.js* se invoca proporcionando el puerto de conexión al cluster y el nombre del fichero a descargar. El programa envía una petición de descarga y queda a la espera de recibir el fichero (o la cadena 'NOT FOUND' si el fichero solicitado no existe en el servidor). Una vez recibe la respuesta, su ejecución concluye.

ACTIVIDAD 3

En esta actividad se ampliará la funcionalidad solicitada en la actividad 2. Para ello se añade el siguiente requisito:

- El master monitorizará el tráfico de mensajes. Cada 5 segundos evaluará el número de peticiones recibidas. Si ese número supera un determinado umbral (4 peticiones servidas por cada worker existente), el master creará un nuevo servidor, usando ***fork()***. Si el número de peticiones no alcanza otro determinado umbral (2 peticiones totales), el master eliminará uno de los servidores, usando ***disconnect()***. En todo momento se garantizará que exista un mínimo de servidores activos (aunque no haya peticiones). Ese valor mínimo será igual al número de procesadores disponibles.

Aparte de realizar esta gestión, el master escribirá en pantalla durante cada evaluación:

- El número de workers activos.
- El número de peticiones atendidas por cada worker en este último intervalo.
- Si va a realizarse alguna acción de escalado (añadir o eliminar servidores) y el tipo de acción que va a realizarse.

ACTIVIDAD 4

En esta actividad se ampliará la funcionalidad solicitada en la actividad 3. Para ello se añade el requisito de que el master escriba en pantalla durante cada evaluación:

- El número de peticiones atendidas por cada worker desde que fue iniciado.
- El número de peticiones atendidas para cada fichero que haya llegado a transferirse, dentro del último intervalo.
- El número de peticiones atendidas para cada fichero que haya llegado a transferirse, desde el inicio.
- El número de peticiones que hayan terminado con error (por ejemplo, por no existir el fichero), desde el inicio.

Para que esto pueda realizarse, se crearán al menos cuatro ficheros que pueda gestionar el servicio de descarga. Además, se extenderá ligeramente la funcionalidad de los 'workers' para soportar estos requisitos.

Debe extenderse también el programa cliente ***zmq-requester.js*** para que solicite de manera automatizada los ficheros existentes (es decir, sin necesidad de que el usuario deba lanzarlo desde la línea de órdenes para realizar cada transferencia). Esta variante podría mantener los nombres de los ficheros a descargar en un vector. Bastaría con pasarle desde la línea de órdenes la duración de la pausa entre cada transferencia solicitada. El programa recorrería el vector de manera circular.

El usuario variaría la carga introducida en el servicio lanzando más o menos instancias del nuevo programa cliente.

ACTIVIDAD 5

Se desea diseñar un **servicio de net** usando el módulo **cluster** de NodeJS. El módulo a implementar deberá cumplir los siguientes requisitos:

1. Lanzar la ejecución de los siguientes procesos: un proceso que sea el cluster master y tantos procesos servidores net como procesadores (cpus) existan en el nodo.

2. Todos los procesos servidores net ejecutarán un servidor como el siguiente:

```
 1: var net = ...
 2: var math = ...
 3:
 4: server = net.createServer(
 5:    function (c){
 6:       c.on('data', function(data){
 7:          var obj = JSON.parse(data);
 8:          var res = math.calculate(obj);
 9:          c.write( obj.func+'('+obj.numb+') = '+res );
10:    });
11: });
```

   Este servidor se proporciona como ejemplo. La resolución de la actividad podría suponer ampliar y/o modificar el código del servidor net a integrar en el cluster. La variable **math** se supone que importa un módulo que calcula funciones matemáticas (Fibonacci y factorial), cuyo código (***auxMath.js***) se incluye al final de este enunciado.

3. Todos los procesos servidores net escucharán peticiones en un mismo puerto, que se proporcionará como argumento en la línea de comandos.

4. Todos los procesos servidores net notificarán al proceso master, mediante mensajes, todas las peticiones de servicio que hayan atendido.

5. El proceso master almacenará contadores de las peticiones atendidas por los servidores net. Para cada servidor, el master deberá guardar el número total de peticiones atendidas y el número de peticiones atendidas en un período reciente (por ejemplo, los últimos 30 o 60 segundos).

6. El proceso master mostrará en consola, periódicamente (por ejemplo, cada 5 o 10 segundos), los contadores de las peticiones atendidas por cada uno de los servidores net.

7. El proceso master, además, monitorizará el rendimiento del servicio, controlando el número promedio de peticiones atendidas por segundo en un intervalo reciente. Este número se quiere mantener en un rango de dos valores [mínimo - máximo]. Dichos valores mínimo y máximo se proporcionarán como argumentos del programa en la línea de comandos[1].

8. El proceso master supervisará periódicamente (por ejemplo, cada 30 o 60 segundos) si el número promedio de peticiones atendidas por servidor se encuentra dentro del

[1] Los valores adecuados de mínimo y máximo dependerán de la máquina donde se ejecute el servicio. Los experimentos que se realicen debieran ayudar en su elección.

rango deseado. En caso de incumplir esta condición, el proceso master tomará las medidas correctoras siguientes:

a. Si el número de peticiones atendidas por servidor y segundo excede el valor máximo deseado, entonces la carga del servicio es elevada y se considera necesario activar un nuevo servidor net. Sin embargo, esto se hará solamente si el número de servidores net activos es menor al doble del número de procesadores (cpus) que existan en el nodo.

b. Si el número de peticiones atendidas por servidor y segundo es menor que el valor mínimo deseado, entonces la carga del servicio es muy baja y se considera necesario eliminar alguno de los servidores net activos. Sin embargo, esto se hará solamente si el número de servidores net activos es mayor al número de procesadores (cpus) que existan en el nodo.

9. El proceso master mostrará en consola un mensaje que informe de que se ha añadido o se ha eliminado algún servidor net, cuando concurran las circunstancias descritas en el punto anterior.

10. Finalmente, el módulo cluster de servidores net que se implemente se deberá probar usando el programa ***net_client.js***, cuyo código se muestra al final de este enunciado. Este programa enviará una petición por milisegundo al módulo cluster.

```
 1: // auxMath.js
 2:
 3: function fibo(n) {
 4:    return (n<2) ? 1 : fibo(n-2) + fibo(n-1)
 5: }
 6:
 7: function fact(n) {
 8:    return (n<2) ? 1 : n * fact(n-1)
 9: }
10:
11: exports.calculate = function (obj) {
12:    var res;
13:    if ( typeof(obj.numb) != 'number' ) res = NaN;
14:    else {
15:       switch (obj.func) {
16:          case 'fibo': res = fibo(obj.numb); break;
17:          case 'fact': res = fact(obj.numb); break;
18:          default: res = NaN;
19:       }
20:    }
21:    return res;
22: }
```

```
 1: // net_client.js
 2:
 3: var net = require('net');
 4:
 5: if ( process.argv.length != 3 ) {
 6:    console.log('uso: node net_client port');
 7:    process.exit(0);
 8: }
 9:
10: function random_request(){
11:    var func = ( Math.random()<0.5 ) ? 'fibo' : 'fact';
```

```
12:    var max  = ( func == 'fibo' ) ? 36 : 100;
13:    var numb = Math.round(Math.random() * max);
14:    return JSON.stringify( {"func":func, "numb":numb} );
15: }
16:
17: function net_client(){
18:    var cont = 0;
19:    var client = net.connect(
20:           {port: process.argv[2]},
21:           function() { //'connect' listener
22:                 client.write( random_request() );
23:    });
24:    client.on('data', function(data) {
25:          console.log( data.toString() );
26:          cont++;
27:          if ( cont<100 )
28:             client.write( random_request() );
29:          else
30:             client.end();
31:    });
32:    client.on('error', function(err) {
33:          client.end();
34:    });
35: }
36:
    setInterval(net_client, 100);
```