

TSR: Actividades del Tema 2, Parte II

ACTIVIDAD 1

OBJETIVO: Aplicar los módulos **net** y **http** de Node.js en el desarrollo de un servicio accesible mediante un navegador web.

ENUNCIADO: Se han de considerar, como punto de partida, los programas vistos en la página 37 de la presentación del tema, cuyos códigos¹ se presentan a continuación:

```
1 // netServer3.js
2
3 const net = require('net')
4 let myF = require('./myFunctions')
5
6 let end_listener = function() { console.log('server disconnected') }
7 let error_listener = function() { console.log('some connection error') }
8 let bound_listener = function() { console.log('server connected') }
9
10 let server = net.createServer(function(c) {
11   c.on('end', end_listener)
12   c.on('error', error_listener)
13   c.on('data', function(data) {
14     let p = JSON.parse(data)
15     let q
16     if (typeof(p.num) != 'number') q = NaN
17     else { switch (p.fun) {
18       case 'fibo': q = myF.fibo(p.num); break
19       case 'fact': q = myF.fact(p.num); break
20       default: q = NaN
21     }}
22     c.write(p.fun+'('+p.num+') = '+q)
23   })
24 })
25
26 server.listen(9000, bound_listener)
```

```
1 // myFunctions.js
2
3 exports.fact = fact
4 exports.fibo = fibo
5 function fact(n) { return (n < 2) ? 1 : n * fact(n - 1) }
6 function fibo(n) { return (n < 2) ? 1 : fibo(n - 2) + fibo(n - 1) }
```

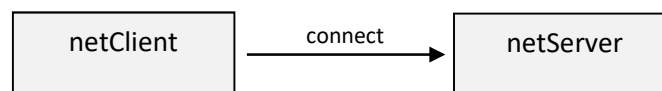
¹ Todos los ficheros fuente listados en este documento se encuentran en un archivo “acts_plus.zip” disponible en PoliformaT.

```

1 // netClient3.js
2
3 const net = require('net')
4
5 if (process.argv.length !== 4) {
6     console.log("Two arguments, function and number, are needed")
7     process.exit()
8 }
9
10 let fun = process.argv[2]
11 let num = Math.abs(parseInt(process.argv[3]))
12
13 // The server is in our same machine.
14 let client = net.connect({port: 9000},
15     function() {
16         console.log('client connected')
17         let request = {"fun":fun, "num":num}
18         client.write(JSON.stringify(request))
19     })
20
21 client.on('data', function(data) {
22     console.log(data.toString())
23     client.end()
24 })
25
26 client.on('end', function() { console.log('client disconnected') })
27 client.on('error', function() { console.log('some connection error') })

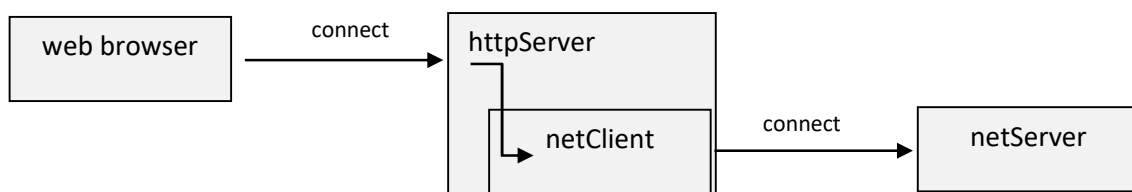
```

Estos programas se pueden representar mediante el siguiente diagrama de componentes:



Se pide modificar el programa *netClient3.js* para que, además de funcionar como cliente del servidor implementado mediante *netServer3.js*, sea un servidor http, de modo que se permita la consulta de las funciones Factorial y Fibonacci (implementadas en *myFunctions.js*) a través de una página web.

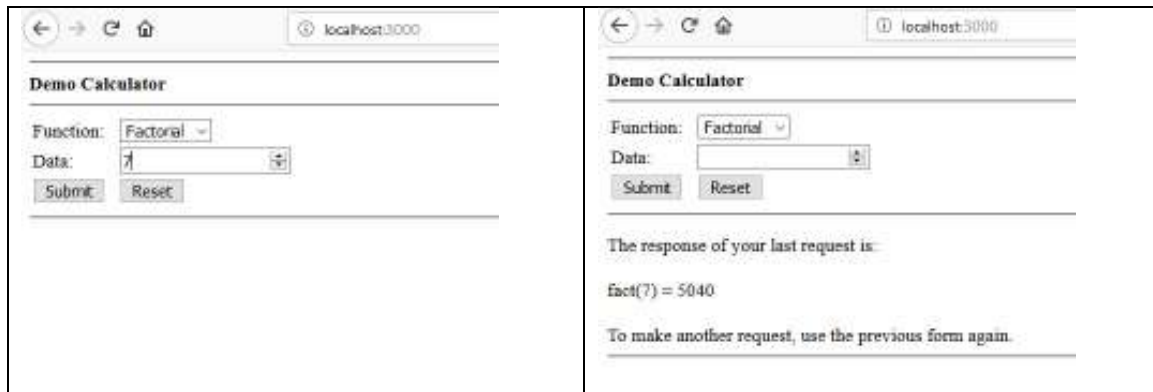
Los programas resultantes se representarían mediante el siguiente diagrama:



El programa que se pide implementar, por tanto, contendrá un servidor http (ofreciendo el servicio a los navegadores web) y un cliente net en un mismo componente. El servidor http recibirá la petición de los usuarios, y redirigirá esa petición, mediante su cliente net al otro

componente, el servidor net. Cuando el servidor net realice el cálculo solicitado, devolverá la respuesta al cliente net y éste, a su vez, devolverá la respuesta al servidor http, que la enviará al navegador web.

Las imágenes en la siguiente tabla muestran el aspecto de la página web antes y después de realizar una consulta para obtener el valor de la función Factorial para el dato 7:



El formulario para la página web se da hecho (en el fichero *formulario.html*), y es el siguiente:

```

1  <form method = "post">
2    <hr>
3    <b>Demo Calculator</b>
4    <hr>
5    <table style="width:20%">
6      <tr>
7        <td>Function:</td>
8        <td><select name="fun">
9          <option value="fact">Factorial</option>
10         <option value="fibo">Fibonacci</option>
11        </select></td>
12      </tr>
13      <tr>
14        <td>Data:</td>
15        <td><input type = "number" name = "num"></td>
16      </tr>
17      <tr>
18        <td><input type = "submit" value = "Submit"></td>
19        <td><input type = "reset" value = "Reset"></td>
20      </tr>
21    </table>
22    <hr>
23  </form>

```

Como ayuda, considerar el siguiente programa incompleto, donde se indica con puntos suspensivos los fragmentos a completar:

```

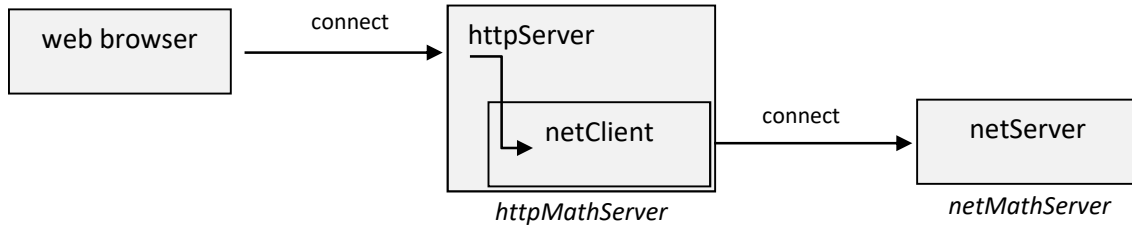
1 // httpMathServer.js (modificación de netClient3.js)
2
3 const http = require('http')
4 const net = require('net')
5 const querystring = require('querystring')
6 const form = require('fs').readFileSync('formulario.html')
7
8 if (process.argv.length !== 4) {
9     console.log("Two arguments, httpPort and netPort, are needed")
10    process.exit()
11 }
12
13 var host = '127.0.0.1'
14 var httpPort = Math.abs(parseInt(process.argv[2]))
15 var netPort = Math.abs(parseInt(process.argv[3]))
16
17 var end_listener = ...
18 var error_listener = ...
19 var data_listener = ...
20
21 var server = http.createServer(function(request, response) {
22     if (request.method === "POST") {
23         var dataPost = ""
24         request.on('data', function(data) { ... })
25         request.on('end', function() {
26             ...
27             let client = net.createConnection({port: netPort}, ... )
28             client.on('end', end_listener)
29             client.on('error', error_listener)
30             client.on('data', data_listener( ... )) // closure
31         })
32     }
33     if (request.method === "GET") {
34         response.writeHead(200, {'Content-Type': 'text/html'})
35         response.end(form)
36     }
37 })
38
39 server.listen(httpPort, hostname, function() {
40     console.log('http server running at http://' + host + ':' + httpPort + '/')
41 })

```

ACTIVIDAD 2

OBJETIVO: Mejorar el rendimiento del servicio desarrollado en la actividad anterior, mediante la replicación del servidor net, y la adición de un nuevo componente: un balanceador de carga.

ENUNCIADO: El servicio desarrollado en la actividad anterior consta de dos componentes, representados en el siguiente diagrama de bloques:

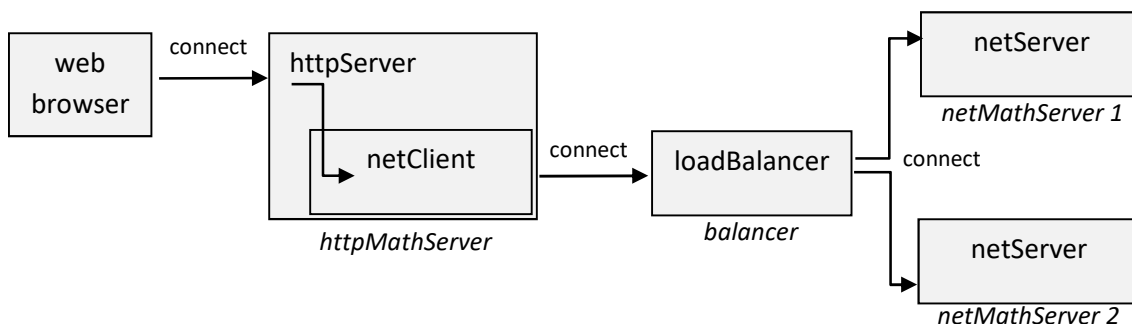


En este servicio, el único servidor net (*netMathServer*) atiende las peticiones secuencialmente, conforme le llegan, y no empieza con una nueva petición hasta que no complete el cálculo de la petición en curso. Si esta petición exige mucho tiempo de cálculo (caso de la sucesión de Fibonacci para valores superiores a 40), entonces también el servidor http (el componente *httpMathServer*) quedará en espera, no pudiendo atender nuevas peticiones ¡de cualquier usuario conectado mediante su navegador web!

La replicación de componentes permite mejorar la calidad del servicio cuando el número de usuarios aumenta. Como una primera aplicación de esta técnica, en esta actividad se propone:

- Instanciar dos réplicas del servidor net (*netMathServer*).
- Implementar un nuevo componente: un balanceador de carga (*balancer*), que discrimine las peticiones según el tiempo de cómputo que impliquen.
- Instanciar una réplica del balanceador de carga (*balancer*).

De esta manera, el diagrama de bloques de los componentes del servicio sería:



El programa, *balancer.js*, que se pide implementar será un componente que contendrá un servidor net (al que se conectará el cliente net del componente *httpMathServer*) y un cliente net (que se conectará a las dos réplicas *netMathServer1* y *netMathServer2*, y repartirá las peticiones recibidas entre esas dos réplicas según un criterio de reparto de carga).

El criterio de reparto de carga será el siguiente:

- a) Las peticiones de cálculo de factoriales, para cualquier valor n , y las peticiones de cálculo de términos de Fibonacci hasta $n = 35$, se dirigirán a una misma réplica (por ejemplo, *netMathServer1*). Esta réplica, por tanto, atenderá solamente peticiones de cómputo ligeras, rápidas.
- b) Las peticiones de cálculo de términos de Fibonacci para $n > 35$, se dirigirán siempre a la otra réplica (por ejemplo, *netMathServer2*). Y así esta réplica se reservará para atender las peticiones de cómputo pesadas, costosas en tiempo.

Cuando el cliente net en *balancer.js* reciba una respuesta de alguna de las réplicas *netMathServer*, pasará internamente esa respuesta a su servidor net. Entonces, este servidor net de *balancer.js* enviará la respuesta al cliente net de *httpMathServer* (componente que se encarga, a su vez, de enviar la respuesta al navegador del usuario).

Como ayuda, considerar el siguiente programa incompleto, donde se indica con puntos suspensivos los fragmentos a completar:

```
1 // balancer.js
2
3 const net = require('net')
4
5 if (process.argv.length !== 5) {
6   console.log("Three arguments ... are needed"); process.exit()
7 }
8
9 let netPort0 = Math.abs(parseInt(process.argv[2]))
10 let netPort1 = Math.abs(parseInt(process.argv[3]))
11 let netPort2 = Math.abs(parseInt(process.argv[4]))
12
13 let server_bound_listener = ...
14 let server_end_listener = ...
15 let client_end_listener = ...
16 let server_error_listener = ...
17 let client_error_listener = ...
18
19 let client_data_listener = function( ... ) {
20   return function(data) { ... }
21 }
22
23 let server_data_listener = function(c) {
24   return function(data) {
25     ...
26     let client = net.createConnection({port: ... }, function() { ... })
27     client.on('end', client_end_listener)
28     client.on('error', client_error_listener)
29     client.on('data', client_data_listener( ... )) // closure
30   }
31 }
32
33 let server = net.createServer(function(c) {
34   c.on('end', server_end_listener)
35   c.on('error', server_error_listener)
36   c.on('data', server_data_listener(c)) // closure
```

```
37  })
38
39  server.listen(netPort0, server_bound_listener)
```

ACTIVIDAD 3

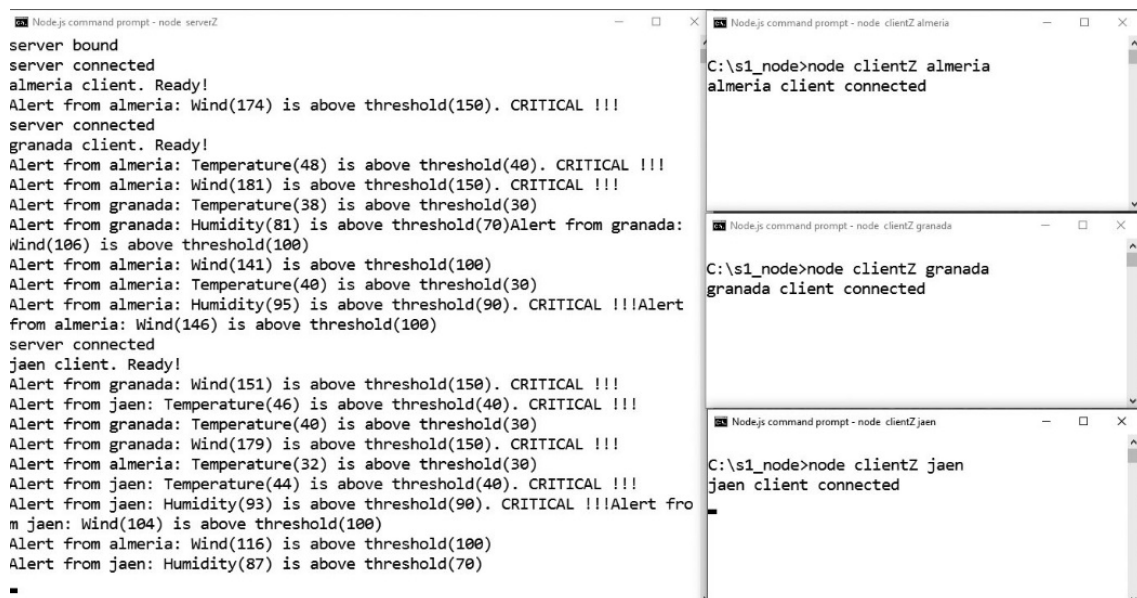
OBJETIVO: Aplicar los módulos **net** y **events** de Node.js en el desarrollo de un servicio de alertas meteorológicas.

ENUNCIADO: Sea un servicio de alertas de tipo meteorológico que se quiere implantar usando el módulo **net**. Cada observatorio, o lugar de mediciones, toma datos acerca de temperatura, humedad y viento. Cada observatorio dispone de un cliente net y este envía mensajes de alerta, a un único servidor net, cuando los valores medidos superan determinados umbrales.

En la simulación de este servicio, la generación de las medidas de temperatura, humedad y viento se hará usando el módulo **events** y generando datos de medida aleatorios.

Se proporciona el programa servidor, y se pide implementar el programa cliente, partiendo de una versión incompleta del mismo, y teniendo en cuenta las indicaciones que se facilitan después de facilitar los códigos fuente.

La siguiente figura muestra un ejemplo de ejecución donde, a la izquierda, se muestra la consola del servidor net, y a la derecha las consolas de tres clientes (supuestamente en Almería, Granada y Jaén):



The screenshot displays four terminal windows. The leftmost window, titled 'Node.js command prompt - node serverZ', shows the server's output: 'server bound', 'server connected', and a series of alerts from three clients (almeria, granada, jaen) indicating when weather data exceeds thresholds (e.g., 'Alert from almeria: Wind(174) is above threshold(150). CRITICAL !!!'). The three windows on the right show client activity: 'Node.js command prompt - node clientZ almeria', 'Node.js command prompt - node clientZ granada', and 'Node.js command prompt - node clientZ jaen'. Each client window shows the command 'C:\s1_node>node clientZ [location]' and the response '[location] client connected'.

El programa servidor², completo, es el siguiente:

```
1 var net = require('net');
2
3 var endListener = function() { console.log('server disconnected'); }
4 var errorListener = function() { console.log('some connection error'); }
5 var dataListener = function(data) { console.log(data.toString()); }
6 var boundListener = function() { console.log('server bound'); }
7
8 var connListener = function(c) { //connection' listener
```

² Disponible en el fichero *serverZ.js*


```

9 console.log('server connected');
10 c.on('end', endListener);
11 c.on('error', errorListener);
12 c.on('data', dataListener);
13 }
14
15 var server = net.createServer(connListener);
16 server.listen(9000, boundListener);

```

El programa cliente³, a completar (se indica con puntos suspensivos los fragmentos a completar), es el siguiente:

```

1  if ( process.argv.length != 3 ) {
2      console.log('uso: node clientZ id'); process.exit(0);
3  }
4
5  var net = require('net');
6  var eve = require('events');
7  var emitter = new eve.EventEmitter();
8
9  // listeners del cliente net
10 var endListener = function() { console.log('client disconnected'); }
11 var errorListener = function() { console.log('some connection error'); }
12 var dataListener = function(data) { console.log(data.toString()); }
13
14 var id = process.argv[2]; // identificador del cliente
15 var evNames = ['Temperature', 'Humidity', 'Wind']; // 3 eventos, en evNames
16
17 // 3 arrays para valores numéricos relativos a eventos
18 // [valor_máximo, valor_umbral_1, valor_umbral_2]
19 var evValues = {}
20 evValues[evNames[0]] = [50, 30, 40]
21 evValues[evNames[1]] = [100, 70, 90]
22 evValues[evNames[2]] = [200, 100, 150]
23
24 var client = net.connect( ... );
25 client.on('data', ... );
26 client.on('end', ... );
27 client.on('error', ... );
28
29 function getValues() { // simulador de datos
30     var a = {};
31     for (var t in evNames) {
32         var s = evNames[t];
33         a[s] = parseInt(Math.random() * evValues[s][0]);
34     }
35     return a;
36 }
37

```

³ Disponible en el fichero *clientZ.js*

```

38 function monitoring() {
39     var a = getValues();
40     ...
41 }
42
43 function reporting(name, value) {
44     ...
45 }
46
47 for (var t in evNames) {
48     var s = evNames[t];
49     ...
50 }
51
52 setInterval( ... , 1000);

```

El programa cliente debe realizar las siguientes acciones:

- Cada segundo ha de ejecutar la función *monitoring*.
- La función *monitoring* simula las mediciones meteorológicas, mediante la invocación de la función *getValues*. Después, para cada evento en *evNames*, por ejemplo *evNames[t]*, debe comprobar si el valor de la medida supera el umbral almacenado en *evValues[evNames[t]][1]*. En tal caso, se debe emitir el evento *evNames[t]* con el valor de su medida.
- Para cada evento debe implementarse un listener. Esto se implementaría en la línea 49 del código anterior a completar. El listener tendrá que invocar a la función *reporting*, pasándole el nombre del evento y el valor de su medida (para hacer esto, será necesaria implementar una clausura).
- La función *reporting* construirá el mensaje de alerta y lo enviará al servidor net (usando la función *write* del cliente net).
- Ejemplos de estos mensajes de alerta se pueden ver en la figura de ejemplo de ejecución mostrada al principio de este enunciado. Por ejemplo, para una alerta de viento desde Almería con un valor de medida 141, el mensaje enviado al servidor net (y que éste mostrará) es: "Alert from almeria: Wind(141) is above threshold(100)".
- La función *reporting* deberá discriminar si el valor de la medida supera *evValues[evNames[t]][2]*, y entonces añadir ". CRITICAL !!!" al mensaje de alerta. Por ejemplo, para una alerta de temperatura desde Almería con un valor de medida 48, el mensaje enviado al servidor net es: "Alert from almeria: Temperature(48) is above threshold(40) . CRITICAL !!!".
- El *connect listener* del cliente net debe mostrar en su consola "... client connected" y enviar a su servidor "...client. Ready!", donde en el lugar de los puntos suspensivos aparecerá el identificador del observatorio, por ejemplo, "almeria".