TSR – 16 octubre 2017. EJERCICIO 1

Escriba en NodeJS dos programas (cliente.js y servidor.js) que satisfagan estos requisitos:

1) Su comunicación debe basarse en *sockets*, utilizando para ello el módulo "net".
2) El programa cliente debe recibir dos argumentos desde la línea de órdenes. El primero especifica cuántos mensajes se enviarán al servidor y el segundo es una cadena a enviar en cada mensaje. Se concatenará el número de mensaje actual a esa cadena.
3) El servidor debe devolver una respuesta (la cadena 'Ok') para cada mensaje recibido. Debe imprimir también el mensaje recibido en la pantalla.
4) El cliente también mostrará cada respuesta recibida en la pantalla.
5) Una vez el cliente haya enviado todos sus mensajes y recibido la última respuesta, cerrará la conexión y finalizará su ejecución.
6) El cliente enviará un mensaje cada segundo.
7) El servidor debe aceptar conexiones desde múltiples clientes y debe utilizar el puerto 9000.
8) Los clientes y el servidor se ejecutan en el mismo ordenador.

Ejemplo de ejecución:

```
$ node servidor &
$ node cliente 3 "Mi mensaje " &
$ node cliente 2 "Otro cliente " &
$ Mi mensaje 1
Ok
Otro cliente 1
Ok
Mi mensaje 2
Ok
Otro cliente 2
Ok
Mi mensaje 3
Ok
```

Referencia básica:

- The `process.argv` property returns an array containing the command line arguments passed when the Node.js process was launched.
- net.createServer([options][, connectionListener]). Creates a new server. The `connectionListener` argument is automatically set as a listener for the `'connection'` event. It returns a net.Server socket object.
- net.connect(port[, host][, connectListener]). A factory function, which returns a new net.Socket and automatically connects to the supplied port and host. If host is omitted, 'localhost' will be assumed. The connectListener parameter will be added as a listener for the 'connect' event once.
- net.Socket.write(data[, encoding][, callback]). Sends data on the socket. The second parameter specifies the encoding in the case of a string--it defaults to UTF8 encoding. Returns true if the entire data was flushed successfully to the kernel buffer. Returns false if all or part of the data was queued in user memory. 'drain' will be emitted when the buffer is again free. The optional callback parameter will be executed when the data is finally written out - this may not be immediately.
- net.Socket.end([data][, encoding]). Half-closes the socket. i.e., it sends a FIN packet. It is possible the server will still send some data. If data is specified, it is equivalent to calling socket.write(data, encoding) followed by socket.end().
- net.Server.listen([port][, hostname][, backlog][, callback]). Begin accepting connections on the specified port and hostname. If the hostname is omitted, the server will accept connections on any IP address. Backlog is the maximum length of the queue of pending connections. The default value of this parameter is 511 (not 512). This function is asynchronous. When the server has been bound, 'listening' event will be emitted. The last parameter callback will be added as a listener for the 'listening' event. One issue some users run into is getting EADDRINUSE errors. This means that another server is already running on the requested port.

SOLUCIÓN

```
// Client.js
const net=require('net');
var socket=net.connect(9000);
var args=process.argv.slice(2);

if (args.length!=2) {
        console.log("Usage: node %s <num-messages> <msg-text>",process.argv[1])
        process.exit(1);
}

const numMsgs=parseInt(args[0]);
const msgText=args[1];
var receivedReplies=0;
var msgCounter=0;

socket.on('data', function(msg) {
        console.log(msg+'');
        receivedReplies++;
        if (receivedReplies==numMsgs) {
                socket.end();
                process.exit(0);
        }

});

setInterval( function () {
        msgCounter++;
        socket.write(msgText+" "+msgCounter);
},1000);
```

```
// Server.js
const net=require('net');
var socket=net.createServer(function (con) {
        con.on('data', function (msg) {
                console.log(msg+'');
                con.write('Ok');
        });
});

socket.listen(9000);
```

TSR – 16 octubre 2017. EJERCICIO 2

Se necesita un programa que reciba desde la línea de órdenes una lista de nombres de fichero. Su salida debe ser el nombre y tamaño del mayor de esos ficheros.

Dos programadores han escrito las siguientes propuestas de solución para ese problema:

```
// Program1.js
const fs=require('fs');
var args=process.argv.slice(2);
var maxName='NONE';
var maxLength=0;
for (var i=0; i<args.length; i++)
        fs.readFile(args[i],'utf8', function(err,data) {
                if (!err) {
                        console.log('Processing %s...',args[i]);
                        if (data.length>maxLength) {
                                maxLength=data.length;
                                maxName=args[i];
                        }
                }
        });
console.log('The longest file is %s and its length is %d bytes.', maxName, maxLength);
```

```
// Program2.js
const fs=require('fs');
var args=process.argv.slice(2);
var maxName='NONE';
var maxLength=0;
function generator(name,pos) {
   return function(err,data) {
        if (!err) {
           console.log('Processing %s...',name);
           if (data.length>maxLength) {
                maxLength=data.length;
                maxName=name;
           }
        }
        if (pos==args.length-1)
           console.log('The longest file is %s and its length is %d bytes.', maxName, maxLength);
   }
}
for (var i=0; i<args.length; i++)
        fs.readFile(args[i],'utf8', generator(args[i],i));
```

Asuma que ambos programas han sido utilizados en un directorio con este contenido:

```
$ ls -go
total 63368
-rw-rw-r-- 1 64880640 Oct  4 15:57 b
-rw-rw-r-- 1      495 Oct  4 16:09 Program1.js
-rw-rw-r-- 1      660 Oct  4 16:10 Program2.js
```

...mediante estas órdenes:

a) node Program1 b Program1.js Program2.js

b) node Program2 b Program1.js Program2.js

Justifique qué salida mostrarán (qué mensajes y en qué orden) esas dos ejecuciones. Ambos programas son sintácticamente correctos y no generan ninguna excepción o error sintáctico durante su ejecución.

SOLUCIÓN

a)

Los mensajes mostrados en pantalla por este primer proceso son:

The longest file is NONE and its length is 0 bytes.
Processing undefined...
Processing undefined...
Processing undefined...

Analicemos por qué se visualizan en ese orden:

- Cada fichero se lee de manera asíncrona. Esto implica que esas lecturas son gestionadas internamente por otros hilos de ejecución. En algún momento cada uno de esos hilos finaliza su lectura y retorna su resultado, ejecutando el *callback* correspondiente. Ese *callback* muestra un mensaje "Processing <nombre-fichero>..."
- Por otra parte, la última línea del programa se ejecuta de manera sincrónica en el hilo principal. Debido a ello, ese mensaje se mostrará en primer lugar. En ese momento, tanto maxName como maxLength todavía tienen sus valores iniciales. Esto explica el contenido de ese primer mensaje: "The longest file is NONE and its length is 0 bytes".
- Posteriormente, cada uno de los *callbacks* es ejecutado. Cuando eso suceda, el bucle "for" utilizado para llamar a fs.readFile() habrá completado todas sus iteraciones. Por tanto, la variable "i" tiene un valor 3 en ese momento. Eso provoca que la sentencia "console.log()" usada en los *callbacks* siempre muestre el mismo nombre de fichero: el contenido de "args[3]". Infortunadamente, el vector "args" únicamente almacena tres nombres en este ejemplo, en las componentes 0 a 2, inclusive. Así, "args[3]" no mantiene ningún valor y su contenido es "undefined". Esa es la información mostrada en los últimos tres mensajes.

b)

Los mensajes que podrían mostrarse en este segundo proceso serían:

Processing Program1.js...
Processing Program2.js...
The longest file is Program2.js and its length is 660 bytes.
Processing b...

Analicemos su contenido y orden:

- En este programa las instrucciones "console.log()" que muestran los mensajes están todas en el *callback* de fs.readFile(). Además, como ese *callback* necesita únicamente dos parámetros (el error y el contenido del fichero, respectivamente) y nosotros queremos utilizar tanto un nombre de fichero (para mostrarlo) como un índice (para compararlo y saber si se debe mostrar el mensaje final), se ha tenido que usar otra función para generar el *callback* y proporcionar la clausura que facilite esos dos datos.
- Con generator() aseguramos que los mensajes "Processing <nombre-fichero>..." muestren un nombre de fichero correcto. Eso implica que en la salida mostrada arriba, los *callbacks* fueron ejecutados en este orden: (1) Program1.js, (2) Program2.js, (3) b. Su orden concreto depende de varios factores: el tamaño de los ficheros y la política de planificación del sistema operativo son los dos principales. El gran tamaño del fichero "b" (al menos si se compara con el tamaño de los demás ficheros) conduce a que se muestre al final. Por otra parte, las posiciones de Program1.js y Program2.js dependen de la política de planificación para las peticiones realizadas al sistema de ficheros. Así, el orden de esos mensajes tiene cierto nivel de incertidumbre, aunque el orden más probable es el que se ha mostrado arriba.
- El mensaje "The longest file..." se mostrará cuando el *callback* corresponda al fichero "Program2.js" (justo después de haber mostrado "Processing Program2.js..."), puesto que eso ocurre cuando el valor del iterador es "args.length-1" (es decir, 2). En nuestra traza no mostrará un resultado correcto (es decir, que el mayor fichero es "b" y su tamaño es 64880640 bytes) porque el *callback* asociado a "b" todavía no ha llegado a ejecutarse.