TSR – 16th October 2017. EXERCISE 1

Please implement in NodeJS two programs (client.js and server.js) with all these requirements:

1) Their communication must be based on sockets, using to this end the ‘net’ module.
2) The client program must receive two command line arguments. The first argument states the amount of messages to be sent to the server, and the second one is a string to be sent in each message. Those messages must append to that string the current message number.
3) The server should return a reply (a message with the ‘Ok’ string) to every received message. It should also print the request message to the screen.
4) The client program should also print the received replies to the screen.
5) Once the intended messages have been sent by the client and their last reply has been delivered, it will close that connection and terminate its execution.
6) The client program sends one message per second.
7) The server program must accept connections from multiple clients and it must be bound to port number 9000.
8) Clients and server run in the same computer.

Execution example:

```
$ node server &
$ node client 3 “My message “ &
$ node client 2 “Another client “ &
$ My message 1
Ok
Another client 1
Ok
My message 2
Ok
Another client 2
Ok
My message 3
Ok
```

Basic reference:

- 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.

SOLUTION

```
// 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 – 16th October 2017. EXERCISE 2

Let us write a program that receives from the command line a list of file names. Its output should be the name and size of the longest program in that set.

Two developers have written the following solution proposals to that problem:

```
// 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));
```

Let us assume that both programs have been used in a directory with the following contents:

```
$ 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
```

...using these commands:

a) node Program1 b Program1.js Program2.js

b) node Program2 b Program1.js Program2.js

Please **justify** the output shown (which messages and in which order) in each of those executions. Note that both programs are syntactically correct and no error or exception will be generated.

SOLUTION

a)

The output generated by that first program is:

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

Let us analyse why those messages are shown in that order:

- Each file is read in an asynchronous way. This means that their callbacks are internally managed by other execution threads. Eventually, each one of those threads terminates the file reading and returns its result, executing the corresponding callback. That callback shows a "Processing <some-file>..." message.
- On the other hand, the last program line is executed synchronously by the main thread. This means that it will be shown first. At that time, maxName and maxLength still have their initial values. That fact explains the first seen message ("The longest file is NONE and its length is 0 bytes").
- Later on, each one of the callbacks is run. At the time they are run, the "for" loop that has been used for calling fs.readFile() has already completed all its iterations. This means that variable "i" has a value of 3 in this example. Therefore, the "console.log()" statement used in those callbacks always prints the same file name: that contained in "args[3]". Unfortunately, the "args" array only stores three file names in this example, corresponding to the slots indexed between 0 and 2, inclusive. Thus, "args[3]" does not hold any value; i.e., its content is "undefined". That is the information printed in the last three messages of this trace.

b)

The output generated by the second program can be:

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

Let us analyse the message contents and order:

- In this program, all the statements that print any information are placed in the callback to be used by fs.readFile(). Besides, since that callback needs only two parameters (the error and the file contents, respectively) and we are interested in printing the correct file name and manage the appropriate array index, we have used another function that generates the callback and provides the needed closure for holding those two other data.
- The usage of the generator() closure ensures that each "Processing <file-name>..." message prints a correct file name. This means that the callbacks that have managed each file have been run in this order in the output example shown above: (1) Program1.js, (2) Program2.js, (3) b. Their concrete order depends on several factors: file length and operating system scheduling strategies, mainly. The very large size of file "b" leads to its last place in that order. On the other hand, the positions for Program1.js and Program2.js depend on the FS request scheduling strategy. Thus, the order of those messages has a non negligible level of uncertainty, although that shown above is the most likely.
- The other message (i.e., "The longest file...") is printed in the callback associated to "Program2.js", since at that time the value of the iterator is equal to "args.length-1" (i.e., 2). Unfortunately, in our assumed output, it is not able to show the intended result (i.e., that the longest file is "b" and its size is 64880640 bytes) because the callback associated to "b" has not terminated yet.