

TSR: Exercises Seminar 1

EXERCISE 1

GOAL: To implement a **net** server that receives electoral results, to be transmitted by means of an indeterminate number of **net** clients. The server will have to accumulate, and save in files all the information that it receives.

The service to be implemented must process the electoral results of some General Elections in Spain. In this case, the electoral district is the **province**, and in each province there is a variable number, but high, of polling places.

The function of each **net** client is to send an object with the votes obtained by each political party in a determinate **polling place**. These are examples of this class of objects:

```
{province:madrid, pollplace:chamberi_14 , pp:3532, psoe:2056, up:3077, cs:1540}
```

```
{province:barcelona, pollplace:sants_12 , pp:105, psoe:143, up:256, cs:196, ERC:238}
```

The objects have always a **province** property that identifies the electoral district, a **pollplace** property that identifies the polling place, and a variable number of properties such that each identifier is that of a political party and its value is the amount of votes obtained by that party.

These objects, serialised with **JSON.stringify**, are the data received by the **net** server. The server has to process the received data, saving them properly in memory and in disk:

- In memory, it has to keep an array, **votes**, that uses the **province** property (of the received objects) as index of the array, and stores in that array slot all the votes received by each party in that electoral district.
- In disk, periodically (every 20 seconds), it has to save a text file (extension ".txt") per slot in the **votes** array. The name of the file will be that of the index of the array (i.e., the **province** property) and the content of the file will be the value stored in that position of the array, serialised with JSON.

As an example, consider that in the first 20 seconds of execution, the server has received the following data from several **net** clients:

```
{province:madrid, pollplace:chamberi_14 , pp:3500, psoe:2000, up:3000, cs:1500}
```

```
{province:barcelona, pollplace:sants_12 , pp:1000, psoe:1500, up:2000, cs:2000, ERC:3000}
```

```
{province:madrid, pollplace:castellana_352 , pp:2000, psoe:3000, up:1000, cs:500}
```

```
{province:valencia, pollplace:vera_sn , pp:2500, psoe:1500, up:2000, cs:2500}
```

```
{province:madrid, pollplace:retiro_8 , pp:4000, psoe:3000, up:2000, cs:2000}
```

```
{province:barcelona, pollplace:provença_115 , psoe:500, up:400, cs:200, erc:300}
```

Then, the variable **votes** should keep the following information:

```
votes['madrid'] = {pp:9500, psoe:8000, up:6000, cs:4000}
```

```
votes['barcelona'] = {pp:1000, psoe:2000, up:2400, cs:2200, erc:3300}
```

```
votes['valencia'] = {pp:2500, psoe:1500, up:2000, cs:2500}
```

And in disk it should have written the following files:

<i>File name</i>	<i>Contents of the file</i>
madrid.txt	{ "pp":9500, "psoe":8000, "up":6000, "cs":4000 }
barcelona.txt	{ "pp":1000, "psoe":2000, "up":2400, "cs":2200, "erc":3300 }
valencia.txt	{ "pp":2500, "psoe":1500, "up":2000, "cs":2500 }

Please implement that **net** server, taking as a base, and keeping in the solution, the following code:

```
1 var net = require('net')
2 var fs = require('fs')
3 var votes = { }
4
5 var server = net.createServer(function(c) {
6     c.on('data', function(data){
7         // To BE COMPLETED
8     })
9 })
10
11 server.listen(9000,
12 function() { console.log('server bound')
13 })
14
15 function save() {
16     // To BE COMPLETED
17     console.log('data has been saved to disk')
18 }
19
20 // To BE COMPLETED
```

EXERCISE 2

GOAL: To implement a program that obtains electoral results reading a group of text files (those written in the previous exercise) and to develop an interactive session (with the user) to consult results of each province.

Let us consider that this program is run in a directory where there are several text files storing the electoral results. The names and contents of the files follow the format described in the previous exercise, like these ones, for instance:

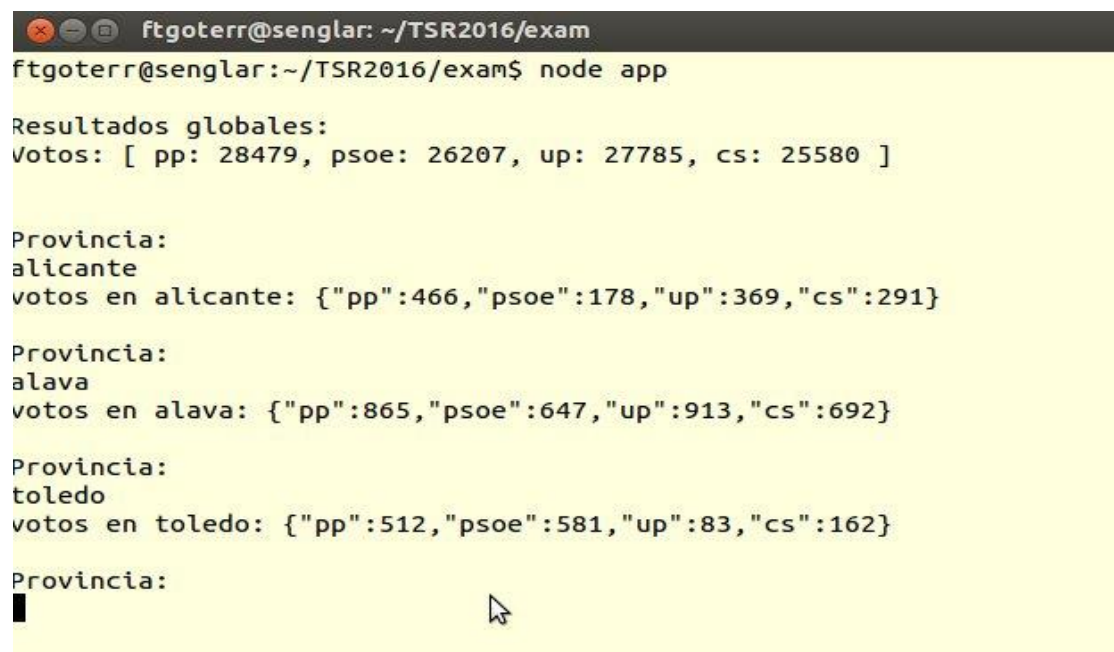
<i>File name</i>	<i>Contents of the file</i>
madrid.txt	{"pp":9500, "psoe":8000, "up":6000, "cs":4000}
barcelona.txt	{"pp":1000, "psoe":2000, "up":2400, "cs":2200, "erc":3300}
Valency.txt	{"pp":2500, "psoe":1500, "up":2000, "cs":2500}

The program, to begin with, has to read all files, keeping the electoral results in an array, **votes**, with the same criterion of the previous exercise.

The program, later (i.e., once all files have been read), has to:

- Show the global results (the total amount of votes obtained by each party in all districts), keeping them in another array called **total_votes**.
- Initiate an interactive session, by means of **process.stdin**, to let the user ask for the results in each province.

The following picture serves as a reference of its functionality:



```
ftgoterr@senglar: ~/TSR2016/exam
ftgoterr@senglar:~/TSR2016/exam$ node app

Resultados globales:
votos: [ pp: 28479, psoe: 26207, up: 27785, cs: 25580 ]

Provincia:
alicante
votos en alicante: {"pp":466,"psoe":178,"up":369,"cs":291}

Provincia:
alava
votos en alava: {"pp":865,"psoe":647,"up":913,"cs":692}

Provincia:
toledo
votos en toledo: {"pp":512,"psoe":581,"up":83,"cs":162}

Provincia:
█
```

In this example, the user has written the names of 3 provinces (*alicante, alava, toledo*), and the rest of its output has been generated by the program to be implemented.

Please implement the requested program, taking as a base, and keeping in the solution, the following code:

```
1 var fs = require('fs')
2 var total_votes = []
3 var votes = []
4
5 fs.readdir('.', function (err, files) {
6   var count = files.length
7   for (var i=0; i < files.length; i++) {
8     // To BE COMPLETED
9   }
10 })
```

Help: Information on the Node.js API

fs.readdir(path[, options], callback)

path <String> | <Buffer>
options <String> | <Object>
 encoding <String> default = 'utf8'
callback <Function>

Asynchronous readdir. Reads the contents of a directory. The callback gets two arguments (err, files) where files is an array of the names of the files in the directory excluding '.' and '..'.

fs.readFile(file[, options], callback)

file <String> | <Buffer> | <Integer> filename or file descriptor
options <Object> | <String>
 encoding <String> | <Null> default = null
 flag <String> default = 'r'
callback <Function>

Asynchronously reads the entire contents of a file. The callback is passed two arguments (err, data), where data is the contents of the file. If no encoding is specified, then the raw buffer is returned. If options is a string, then it specifies the encoding.

fs.writeFileSync(file, data[, options])

file <String> | <Buffer> | <Integer> filename or file descriptor
data <String> | <Buffer>
options <Object> | <String>
 encoding <String> | <Null> default = 'utf8'
 mode <Integer> default = 0o666
 flag <String> default = 'w'

The synchronous version of fs.writeFile(). Returns undefined.

net.createServer([options][, connectionListener])

Creates a new server. The connectionListener argument is automatically set as a listener for the 'connection' event.

net.Socket

This object is an abstraction of a TCP or local socket. net.Socket instances implement a duplex Stream interface. They can be created by the user and used as a client (with connect()) or they can be created by Node.js and passed to the user through the 'connection' event of a server.

Event: 'data'

Emitted when data is received. The argument data will be a Buffer or String. Encoding of data is set by socket.setEncoding(). Note that the data will be lost if there is no listener when a Socket emits a 'data' event.

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.

The optional callback parameter will be executed when the data is finally written out - this may not be immediately.

process.stdin

The process.stdin property returns a ***Readable stream*** equivalent to or associated with stdin.

stream.Readable

Event: '**data**'

The 'data' event is emitted whenever the stream is relinquishing ownership of a chunk of data to a consumer.

The listener callback will be passed the chunk of data as a string if a default encoding has been specified for the stream using the readable.setEncoding() method; otherwise the data will be passed as a Buffer.
