NIST - LAB 1 COURSE 2019/20

INTRODUCTION TO THE LABS: JAVASCRIPT AND NODEJS

This lab consists of three sessions, with the following goals:

1. Present the needed tools and procedures to carry out the lab work for NIST.
2. Present the basic techniques for software development using javascript on NodeJS.

It is assumed that you have gone through the material made available to you as "**Lab 0**". In addition, you also have access to reference material describing the general resources available as part of the DSIC labs, as it concerns NIST.

INTRODUCTION

Tools

All lab programming work will be carried out using JavaScript on NodejS.

The computers available at the lab room use shared virtual machine environments. The consequence is that several resources, like **network ports**, are shared among various users, making the environment unsuitable for carrying out the exercises of NIST (with heavy usage of network interactions).

In order to avoid conflicts, the network port numbers mentioned in this document must be adapted by the students. Ports in the range 50000 to 59999 are reserved for the NIST labs. Since no exercise will need more than 10 ports, the least significant digit of those 10 ports must range from 0 to 9. The next three least significant digits will coincide with the three least significant digits of the student's ID card (DNI or passport).

Thus, in any given exercise, a student with DNI 29332481 should replace ports 8000, 8001,…,8009, with ports 54810, 54811,…,54819 respectively.

The main characteristics of the shared virtual machine environments you will have acces to are:

- LINUX (64-bit CentOS7 64), with access to your own account home directory.
- NodeJS (version 10) environment.
- Basic auxiliary libraries (fs, http, net, ...)
- NodeJS npm package management tool
- We can use the pre-installed code editors (Visual Studio Code, available through the Programming menu).

Students can install similarly-capable environments (from the point of view of NodeJS and tools) on their own computers (Windows, LINUX o MacOS), check http://nodejs.org.

Procedures

Clause of good use of lab resources

Resources are made available to students only for the purpose of supporting the educational goal of the school. A correct utilization of them is the exclusive responsibility if the student. Consequently, students commit to:

- Treat their passwords as confidential information, avoiding sharing them.
- Avoid interfering with the activity of other students.
- Use the resources for ends other tan those presented in the lab bulletins.

Our installations contain resource monitoring mechanisms that can be queried for usage information when needed.

JavaScript programs are text files with a .js extension. A program written in file **x.js** is executed with command **node x[.js] [optionalArgs]**

The three sessions of Lab 1 are organized in two parts, the first one focusing on JavaScript (the language); while the second focuses on the NodeJS environment.

As you proceed through the exercises presented to you it will be normal to encounter some errors. The most frequent ones are summarized in the following table.

Syntax errors		
Definition	Detection/solution	Example
Illegal syntax (invalid identifier, structures incorrectly nested, etc.)	The interpreter indicates the error type and its location in the code. Correct on code source	> "Potato"(); TypeError: string is not a function at repl:1:9 at REPLServer.defaultEval (repl.js:132:27) at bound (domain.js:254:14) at REPLServer.runBound [as eval] (domain.js:267:12) at REPLServer.<anonymous> (repl.js:279:12)
Errors in the logic		
Definition	Detection/solution	Example
Programming error (try to access a non-existent property or 'undefined', pass arguments of an incorrect data type, etc.) JavaScript is not strongly type-oriented: some errors, which other languages catch at compilation time, are only shown when the program is run.	Wrong results when executing. Modify the code. It suits that in the code verify always the restrictions to apply on the arguments of the functions	> function add(array) {return array.reduce(function(x,y){return x+y})} undefined > add([1,2,3]) 6 > add(1) TypeError: undefined is not a function at add (repl:1:36) at repl:1:1 at REPLServer.defaultEval (repl.js:132:27) at bound (domain.js:254:14) at REPLServer.runBound [as eval] (domain.js:267:12)

Operational error	
Definition	Detection/solution
Exceptional situations that can arise during the execution of the program. They can be due to: • The environment (e.g., memory exhaustion, too many open files) • Current system configuration (e.g., there is no route to a remote host) • Use of the network (e.g., problems with the use of sockets) • Access problems to a remote service (e.g., I cannot connect to the server) • Wrong or inconsistent input data • Etc.	Use of **try**, **catch**, and **throw**, similar to those of Java Strategy: - If it is clear how to resolve the error, manage it (e.g., error when opening a file for writing -> create it as a new file) - If the management of the error is not responsibility of this fragment of the program, propagate the error to the caller. - For transitory errors (e.g., network-related problems), retry that operation If we cannot manage neither propagate the error: - If it prevents the program from going on, abort it. - In another case, write the error down in a log file.

In order to minimize errors, we recommend:

- Use the strict mode: **`node --use_strict file.js`**
- Document correctly each function
  - Meaning and type of each argument, as well as any additional constraint
  - Which type of operational errors can appear, and how to manage them

- Its intended return value

Part One

Running JavaScript Programs

Attached to the documentation prepared for this lab, you can find a folder with name `javascript` containing several simple programs illustrating various aspects of the JavaScript language, required for NIST.

- These programs illustrate some basic characteristics of JavaScript, such as the ***this*** object reference, the ***bind*** builtin function method*, closures, asynchronous programming,* etc
- A list of the available programs can be found at the end of this bulletin.

The activity for this part consists in the study, analysis and execution of those programs, reaching the pertinent conclusions.

Your professor can direct you through those codes he/she deems more appropriate to emphasize.

Part Two

Introduction to NodeJS

Accessing files

All methods related to files appear in the 'fs' module. The operations are asynchronous by default. However, for each asynchronous function **xx**, there is also a synchronous variant **xxSync**. The following pieces of code can be directly copied to a js file and executed.

- Read asynchronously the contents of a file (`read1.js`)

```
const fs = require('fs');
fs.readFile('/etc/hosts', 'utf8', function (err,data) {
  if (err) {
    return console.log(err);
  }
  console.log(data);
});
```

- Write asynchronously content into a file (`write1.js`)

```
const fs = require('fs');
fs.writeFile('/tmp/f', 'contenido del nuevo fichero', 'utf8',
  function (err) {
```

```
  if (err) {
    return console.log(err);
  }
  console.log('se ha completado la escritura');
});
```

- Modifications on reads and writes: using modules.

We define module **`fiSys.js`**, based on **`fs.js`**, to access files, along with some usage examples.

(**`fiSys.js`**)

```
//Module fiSys
//Ejemplo de módulo de funciones adaptadas para el uso de ficheros.
//(Podrían haberse definido más funciones.)

const fs=require("fs");

function readFile(fichero,callbackError,callbackLectura){
	fs.readFile(fichero,"utf8",function(error,datos){
		if(error) callbackError(fichero);
			else callbackLectura(datos);

	});
}


function readFileSync(fichero){
	var resultado; //retornará undefined si ocurre algún error en la lectura
	try{
		resultado=fs.readFileSync(fichero,"utf8");
	}catch(e){};
	return resultado;
}


function writeFile(fichero,datos,callbackError,callbackEscritura){
	fs.writeFile(fichero,datos,function(error){
			if(error) callbackError(fichero);
				else callbackEscritura(fichero);
	});
}

exports.readFile=readFile;
exports.readFileSync=readFileSync;
exports.writeFile=writeFile;
```

(**read2.js**)

```
//Lecturas de ficheros

const fiSys=require("./fiSys");


//Para la lectura asíncrona:
console.log("Invocación lectura asíncrona");
fiSys.readFile("/proc/loadavg",cbError,formato);
console.log("Lectura asíncrona invocada\n\n");

//Lectura síncrona
console.log("Invocación lectura síncrona");
const datos=fiSys.readFileSync("/proc/loadavg");
if(datos!=undefined)formato(datos);
        else console.log(datos);
console.log("Lectura síncrona finalizada\n\n");

//- - - - - - - - - - -
function formato(datos){
        const separador=" "; //espacio
        const tokens = datos.toString().split(separador);
        const min1 = parseFloat(tokens[0])+0.01;
        const min5 = parseFloat(tokens[1])+0.01;
        const min15 = parseFloat(tokens[2])+0.01;
        const resultado=min1*10+min5*2+min15;
        console.log(resultado);
}

function cbError(fichero){
        console.log("ERROR DE LECTURA en "+fichero);
}
```

(**write2.js**)

```
//Escritura asíncrona de ficheros

const fiSys = require('./fiSys');

fiSys.writeFile('texto.txt','contenido del nuevo fichero',cbError,cbEscritura);

function cbEscritura(fichero){
        console.log("escritura realizada en: "+fichero);
}

function cbError(fichero){
		console.log("ERROR DE ESCRITURA en "+fichero);
}
```

Asynchronous programming: customized events

Usage of the **events** module.

(`emisor1.js`)

```
//Event emitter

const ev = require('events');
const emitter = new ev.EventEmitter();
const e1 = "print", e2= "read";   // name of  events
var n1 = 0, n2 = 0;         // auxiliary vars

// register listener functions on the event emitter
emitter.on(e1,
   function() {
      console.log('event '+e1+' '+n1+++' times')})
emitter.on(e2,
   function() {
      console.log('event '+e2+' '+n2+++' times')})

emitter.on(e1,  // more than one listener for the same event is possible
   function() {
     console.log('something has been printed!')})

//The event generation and propagation are synchronous

emitter.emit(e1);
emitter.emit(e2);
console.log("-----------------------------------------------------------");

//The generation of events can be done asynchronous, for example,
//by means of setTimeout or setInterval

// Generate the events periodically
setInterval(
   function() {emitter.emit(e1);}, // generates e1
   2000);                      // every 2 seconds

setInterval(
   function() {emitter.emit(e2);}, // generates e2
   8000);                      // every 8 seconds

console.log("\n\t===========> end of code: events become emitted periodically\n");
```

Custom module **`generadorEventos.js`** , and its usage.

```
//Módulo generadorEventos

const EventEmitter = require('events').EventEmitter;

const emisor=new EventEmitter();

function Evento(evento,entidadEmisora,valor){
  return {
            emit:function(incr){
                    valor += incr;
                    emisor.emit(evento,entidadEmisora,evento,valor);
            },
           on:function(listener){
                    emisor.on(evento,listener);
            }
  }
}
module.exports=Evento;
```

(**`emisor2.js`**)

```
//Emisor de eventos

const Evento=require("./generadorEventos");

function visualizar(entidad,evento,dato){
  console.log(entidad,evento+'··> ',dato);
}

const evento1 = 'e1';
const evento2 = 'e2';

const emisor1 = Evento(evento1,'emisor1:   ',0);
const emisor2 = Evento(evento2,'emisor2:   ','');

emisor1.on(visualizar);
emisor2.on(visualizar);

console.log('\n\n------->> inicial\n\n');

for(var i=1;i<=3;i++) emisor1.emit(i);

console.log('\n\n------------------>> intermedio\n\n');

for(var i=1;i<=3;i++) emisor2.emit(i);
```

```
console.log('\n\n-------------------------------->> fin\n\n');
```

Activity: Given the following code fragment

(**`emisor3.js`**)

```
//Event emitter

const Evento=require("./generadorEventos");

const evento1 = 'e1';
const evento2 = 'e2';
var incremento=0;

const emisor1 = Evento(evento1,'emisor1:   ',0);
const emisor2 = Evento(evento2,'emisor2:   ','');

function visualizar(entidad,evento,dato){
   console.log(entidad,evento+'··> ',dato);
}

emisor1.on(visualizar);
emisor2.on(visualizar);


emisiones();

//. . . . . . . . . . . . . .
```

Complete **`emisor3.js`** implementing function **`emisiones()`**, and whatever additional code deemed necessary, such that events 1 and 2 are alternatively emitted periodically, such that the following conditions are met:

1) At the end of each period two events are generated: one event **`evento1`** and one event **`evento2`**, passing the value of variable **`incremento`**.
2) At the end of each period, after the events have been generated, variable **`incremento`** will increase by 1.
3) Each new period will have a randomly chosen duration between 2 and 5 seconds.
4) At the end of each period, its duration must be shown on the console.

Client/server interaction

HTTP MODULE

To develop web servers (i.e., HTTP servers)

E.g.- simple web server (`ejemploSencillo.js`) greeting users contacting the server

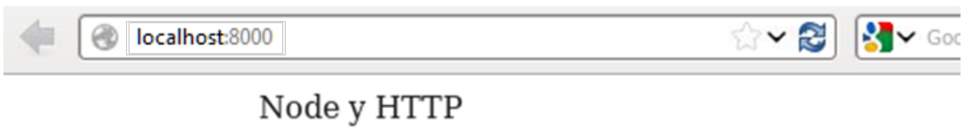
Code	Comments
`const http = require('http');`	It imports module http
	dd(8) -> "08"
`function dd(i) {return (i<10?"0":"")+i;}`	dd(16) -> "16"
`const server = http.createServer(`	It creates the server and
`   function (req,res) {`	associates it this
`      res.writeHead(200,{'Content-Type':'text/html'});`	function
`      res.end('<marquee>Node y Http</marquee>');`	that returns a fixed
`      var d = new Date();`	response
`      console.log('alguien ha accedido a las '+`	and
`         d.getHours() +":"+`	writes the current time
`         dd(d.getMinutes()) +":"+`	to the console
`         dd(d.getSeconds()));`	
`}).listen(8000);`	The server listens to port
	8000

This is the first example of program that uses network ports. Therefore, you should consider that port conflicts may arise and, because of this, port numbers should be adapted. Let us assume that your DNI is 29332481; so, you may use ports 54810 to 54819. Thus, the port number shown in this program could be replaced by the first one available in your range: 54810.

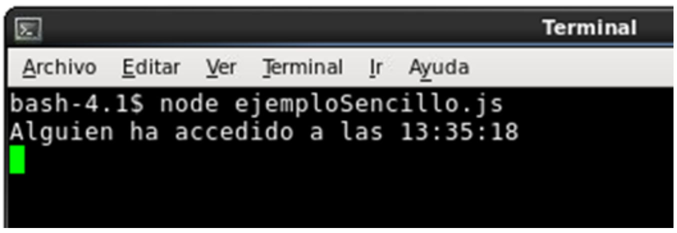
Please, run the server and use a web browser as its client:

- Access to the URL http://localhost:8000 (i.e., http://localhost:54810 in the example discussed above)

- Check (in the browser) which is the response from the server, and in the console which is the message written by the server.



Node y HTTP



```
bash-4.1$ node ejemploSencillo.js
Alguien ha accedido a las 13:35:18
```

NET MODULE

Client (`netClient.js`)

```
const net = require('net');

const client = net.connect({port:8000},
    function() { //connect listener
        console.log('client connected');
        client.write('world!\r\n');
    });

client.on('data',
    function(data) {
        console.log(data.toString());
        client.end(); //no more data written to the stream
    });

client.on('end',
    function() {
        console.log('client disconnected');
    });
```

Server (`netServer.js`)

```
const net = require('net');

const server = net.createServer(
    function(c) { //connection listener
        console.log('server: client connected');
        c.on('end',
            function() {
                console.log('server: client disconnected');
            });
        c.on('data',
            function(data) {
                c.write('Hello\r\n'+ data.toString()); // send resp
                c.end(); // close socket
            });
    });

server.listen(8000,
    function() { //listening listener
        console.log('server bound');
    });
```

Accessing CLI arguments

When launching a program through the CLI, we are interacting with a Shell program. This Shell gathers all arguments passed to the execution command and passes them to the JS program, packed within an array which can be accessed with **process.argv** (e.g., attribute argv of object **process**). We can then count the number of arguments (length of the array) and access each argument positionally (index into the array):

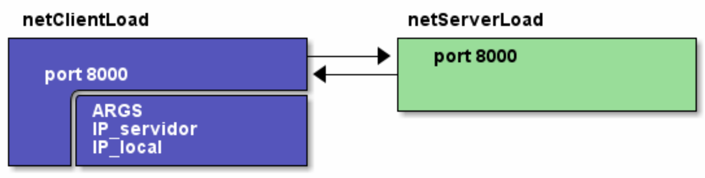
- `process.argv.length:` number of arguments on the CLI
- `process.argv[i]:` get i-th argument.
  - If we used command "**`node program arg1`** …" then `process.argv[0]` contains string `'node'`, `process.argv[1]` contains string `'program'`, `process.argv[2]` string `'arg1'`, etc.

NOTE: you can use **process.argv.slice(2)** to obtain an array which has discarded 'node' and the program path, so that you are left with just the actual arguments passed to the JS program.

Query a computer load

Some scalable client/server systems, replicate their servers (using horizontal scalability), and deliver the requests among them according to their current workload level.

We create the embryo for a system of this type, with a single client and a single server (possibly on different machines) communicating through port 8000



- The server is called **netServerLoad.js**, and it does not receive command-line arguments.
  - The following function **getLoad** computes its current load. That function reads the data from the file **/proc/loadavg**, filters its interesting values (adds one hundredth of a second to them in order to avoid the confusion between value 0 and an error), and processes them calculating a weighted average (with weight 10 to the load of the last minute, weight 2 to the last 5 minutes, and weight 1 to the last 15)

```
   function getLoad(){
     data=fs.readFileSync("/proc/loadavg"); //requiere fs
     var tokens = data.toString().split(' ');
     var min1  = parseFloat(tokens[0])+0.01;
     var min5  = parseFloat(tokens[1])+0.01;
     var min15 = parseFloat(tokens[2])+0.01;
     return min1*10+min5*2+min15;
  };
```

- The client file is **netClientLoad.js**: it receives as its command-line arguments the IP address of the server and its local IP address.
- Protocol: When the client sends a request to the server, it includes its own IP, the server computes its load and returns a response to the client that includes its own IP (i.e., that of the server) and the workload level (i.e., the result of the call to **getLoad**).

To help you test and debug your code, we are keeping an active server at **tsr1.dsic.upv.es**

Important:

- Your starting point to develop these programs is the code for netClient.js and netServer.js seen earlier
- You must make sure that all processes terminate (e.g. using `process.exit()`)
- You can use either the domain name or the IP address for a server machine. Use the **ip addr** command to get the local IP address.
- Complete both programs, place them in different computers (with the help of a classmate), and test their communication through port 8000: `netServerLoad` must

compute the load as an answer to each request from a client. Correspondingly, `netClientLoad` must use its console to show the answer received from the server.

Transparent proxy

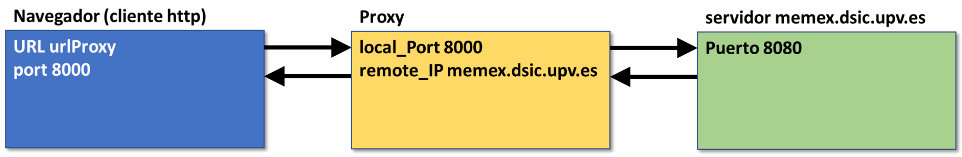
A proxy is a server that, when invoked by a client, forwards the request to a third party (the real server), who, after processing the request, returns it to the proxy, which, in turn routes it to the original client that sent the request.

- From a client's point of view, it works as the real server (hides the real server)
- It can run on an arbitrary machine (unrelated to the client's or server's)
- The proxy can use different ports/addresses, but does not modify the messages in transit.

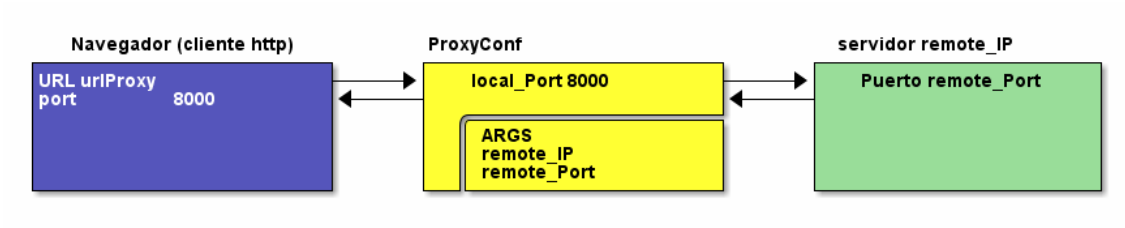
More info in http://en.wikipedia.org/wiki/Proxy_server

You should progress through three incremental stages:

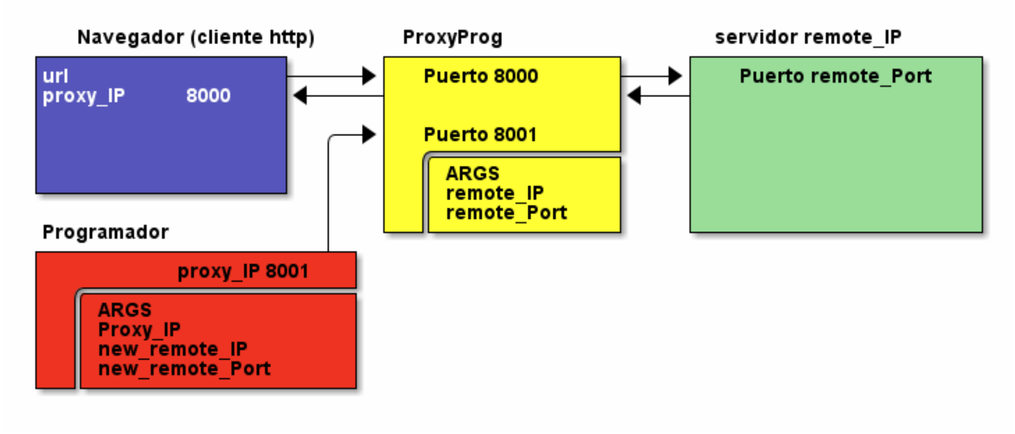
1. Basic proxy (**Proxy**). When an HTTP client (e.g., a browser) contacts proxy's port 8000, the proxy forwards that request to the "memex" web server (158.42.179.56 port 8080), returning each server answer to the client



2. Configurable Proxy (**ProxyConf**): instead of hard coding the IP and port of the real target server in its code, it gets them through command line arguments



3. Programmable Proxy (**ProxyProg**). Initially, the IP address and port number are passed through command line arguments, but they can be altered at run time, sending an appropriate message to port 8001 of the proxy.



Important:

- We provide the code for Proxy.js (although with different IP address and port number, to be changed by you): you should study it until you understand how it works.
    - Verify it works correctly by pointing a browser to URL http://proxy_address:8000/

Code (Proxy.js)

```
const net = require('net');

const LOCAL_PORT  = 8000;
const LOCAL_IP  = '127.0.0.1';
const REMOTE_PORT = 80;
const REMOTE_IP = '158.42.4.23'; // www.upv.es

const server = net.createServer(function (socket) {
     const serviceSocket = new net.Socket();
     serviceSocket.connect(parseInt(REMOTE_PORT),
        REMOTE_IP, function () {
         socket.on('data', function (msg) {
              serviceSocket.write(msg);
         });
         serviceSocket.on('data', function (data) {
            socket.write(data);
         });
     });
}).listen(LOCAL_PORT, LOCAL_IP);
console.log("TCP server accepting connection on port: " +
LOCAL_PORT);
```

Comments

Uses a socket to talk with the client (socket) and another to talk with the server (serviceSocket)

1.- It opens a connection to the server
2.- It reads a message (`msg`) from client
3.- It writes a copy of the message
4.- It waits for an answer, and copies it to the client.

- In scenario 2, you must test the following cases:
    - Access to **UPV**'s server should still work
    - ProxyConf proxying between netClientLoad and netServerLoad
        - If ProxyConf runs in the same machine as netServerLoad, their ports will collide -> modify netServerLoad's code to use a different port.
        - If you get an "EADDRINUSE" error when running the proxy, another program is using the proxy's port.
- In scenario 3 we need to code the program used to reconfigure the proxy. We call it **programador.js**
    - **programador.js** gets the IP of the proxy, as well as the IP and port of the new target server through command line arguments.
    - **programador.js** encodes the new server's IP and port, and sends them as a message to port 8001 of the proxy's machine, after which it terminates.
    - The format of **programador.js** messages should be like this:

```
var msg = JSON.stringify ({'remote_ip':"158.42.4.23", 'remote_port':80})
```

You can test your program using as target servers those already mentioned in this bulletin (memex.dsic.upv.es, IP 158.42.179.56, port 8080; and www.upv.es, IP 158.42.4.23, port 80) or any other that uses HTTP (instead of HTTPS), like www.alboraya.es (IP 62.81.165.166, port 80) or www.villena.es (IP 213.0.62.51, port 80).

REFERENCES

1. `Lab 0`, in PoliformaT, within the *resources* tab for NIST
2. `Laboratorios del DSIC`, also within the *resources* tab for NIST, in PoliformaT
3. List of examples within `javascript` folder

`j00.js`	Functions, objects, reference ***this*** and function method ***bind***().
`j01.js`	Variable declaration. Using functions and closures.
`j02.js`, `j03.js`	Closures with variables and functions.
`j04.js`, `j05.js`, `j06.js`, `j07.js`, `j08.js`, `j09.js`, `j10.js`, `j11.js`	Asynchronous operations, using function *setTimeout*.
`j12.js`	Summary exercise (async operations and closures)
`j13.js`	Fork-join like execution of concurrent async operations.
`j14.js`	Serialized execution of async operations.