Programación Distribuida 2016

Tema: Desarrollo distribuido en NodeJS

Alumno: Acosta Martín - 8502/6

Fecha: 10/02/2016

API

Índice

Presentación

Partiendo de la definición que se ofrece en el sitio de Node.js:

“Node.js es una plataforma construida encima del entorno de ejecución javascript de Chrome para fácilmente construir rápidas, escalables aplicaciones de red. Node.js usa un modelo de E/S no bloqueante dirigido por eventos que lo hace li-gero y eficiente, perfecto para aplicaciones data-intensive en tiempo real”

En efecto, Node.js provee un entorno de ejecución para un determinado lenguaje de programación y un conjunto de librerías básicas, o módulos nativos, a partir de las cuales crear aplicaciones orientadas principalmente a las redes de comunicación.
El potencial para el desarrollo de aplicaciones distribuidas en Node.js se encuentra en la característica mencionada anteriormente. Node.js fue creado desde el principio para orientarse al desarrollo de aplicaciones orientadas a la comunicación en una red. Si a eso sumamos su concurrencia orientadas a eventos obtenemos una plataforma lo suficientemente apta como para diseñar toda clase de modelos de distribución como los practicados en la cursada.

Me pareció necesario introducir brevemente qué es Node.js porque a partir de ello quiero mostrar que existen facilidades que nos da dicha plataforma y JavaScript para construir diferentes tipos de aplicaciones distribuidas.

Javascript, odiado por muchos (y con absoluta razón en varios aspectos), no deja de destacarse por ciertas características como:

- Closures: parte vital del paradigma de Programación Funcional .
- Orientación a prototipos.
- Funciones Lambda.

y si eso le sumamos un entorno como Node.js que posee interesantes características como su soporte de red en los módulos “built-in” o su ecosistema.
Dentro de este ecosistema encontramos por ejemplo la filosofía de combinar módulos (como en Unix) y es así que encontramos distintos módulos capaces de ofrecernos construir la aplicación distribuida que buscamos.

Programación distribuida con el módulo built-in: net

De por sí Node.js con su modulo net nos ofrece un mecanismo inicial/sencillo para programar en sistemas distribuidos, los conocidos sockets.

“net” proporciona al programador el objeto Socket, que representa obviamente un socket TCP o un socket de dominio UNIX , y el objeto Server, que modela un servidor de conexiones TCP. Son las dos perspectivas desde las que se tratan las conexiones TCP dentro de Node.js.

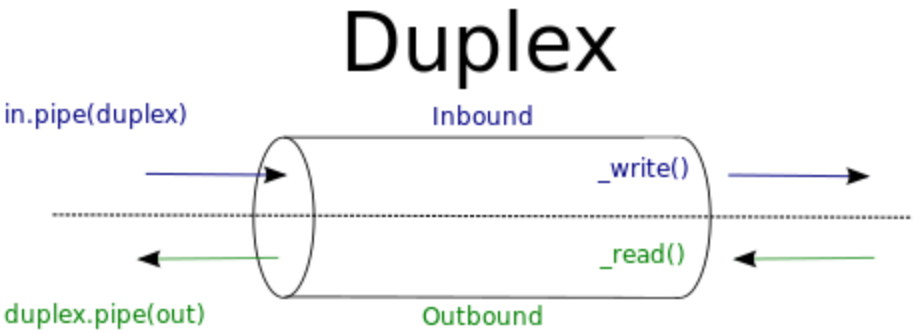
El intercambio de información se realiza a través de operaciones de lectura y escritura sobre el flujo de datos que el socket es, ya que implementa la clase Stream.

Un **Stream** en Node.js es un objeto que encapsula un flujo binario de datos y provee mecanismos para la escritura y/o lectura en él. Por tanto, pueden ser readable, writable o ambas cosas.

En realidad es una interfaz abstracta, con lo cual condiciona a cualquier objeto que quiera comportarse como un Stream a heredar de él y a ofrecer una serie de métodos para el manejo de dicho Stream y, además, condiciona al programador a implementar estos métodos de manera que sean capaces de realizar operaciones sobre el flujo concreto que yace por debajo. Esto se debe a que cada flujo binario puede tratarse de diferentes maneras y las acciones de lectura/escritura no tienen porqué ser iguales a las de otro flujo que también las ofrezca.

Por ejemplo, una conexión TCP y un archivo del sistema de ficheros en Node.js se tratan como Streams, sin embargo, la implementación de las operaciones sobre ellos son distintas por la propia naturaleza de cada uno: datos sobre un protocolo de red vs. datos almacenados en el sistema de ficheros del sistema operativo.

Existe una clase de stream llamada Duplex Stream y que es utilizada por una libreria mas adelante en los ejercicios.
Los Duplex Stream en términos generales son canales bidireccionales. Internamente lo que se tiene son dos stream independientes, out e in. El mejor ejemplo en accion de ellos son los sockets de Node.js.



El módulo **net** es uno de los componentes más importantes dentro de Node.js. Con el, podemos crear desde un simple servidor http hasta un completo sistema de streaming y mucho más mediante módulos de terceros como **dnode** que te permite realizar sistemas con rpc asincrónico e inclusive bajo una visión distinta conseguir compartir interfaces de objetos como ofrece RMI en Java.

Los callback en javascript y la concurrencia I/O en Node.js

Antes de avanzar con los ejercicios quiero explicar algo que utilizo constantemente en mis scripts, los **callback**.
Como la palabra en inglés lo indica un callback es una "llamada de vuelta" y este es un concepto importante al momento de escribir código. Es simple: llamo a una función y le envío por parámetro otra función (un callback) esperando que la función que llamé se encargue de ejecutar esa función callback.

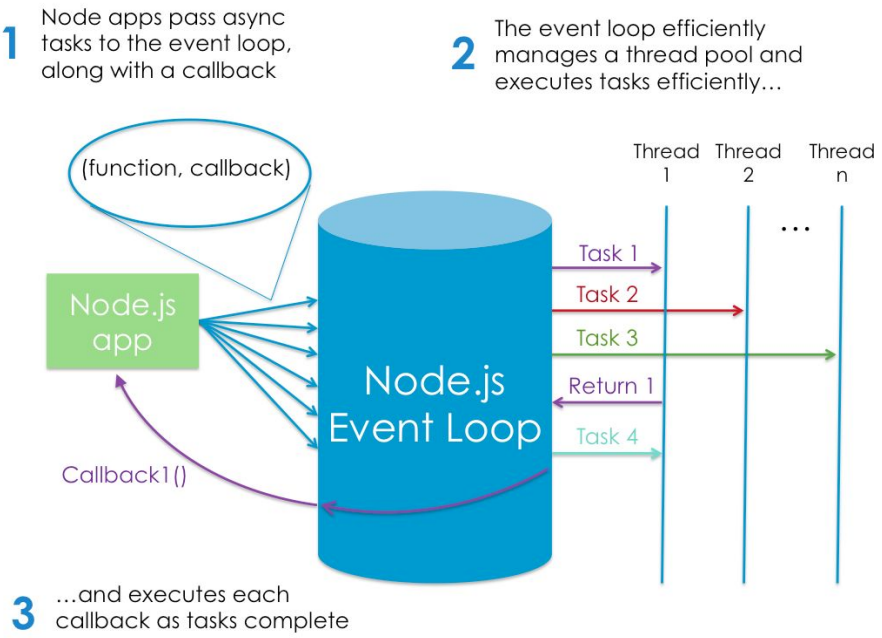
```
function haceAlgo(miCallback){
    //hago algo y llamo al callback avisando que terminé
    miCallback();
}

haceAlgo(function(){
   console.log('terminó de hacer algo');
});
```

Ahora, imagine que tenemos una función que lee un archivo (operación I/O).
En Node.js esta operación la podemos hacer de manera asincrónica y de esta forma nuestro programa podría continuar su ejecución hasta que se ejecute el callback de retorno que le pasamos como parámetro a la función de lectura.

```
var fs = require("fs");
// funcion callback
var miCallback = function(error, data) {
  console.log('Me devolvieron la info del archivo');
};
fs.readFile("foo.txt", miCallback);
console.log('Mientras sigo mi ejecución');
```

Output:
>> Mientras sigo mi ejecución
>> Me devolvieron la info del archivo

Entonces ¿cómo puede ser que Node.js el cual se ejecuta sobre un solo thread pueda tener esta característica no bloqueante con operaciones I/O?
La respuesta es que en Node.js tenemos concurrencia en las operaciones I/O.
Volviendo al ejemplo anterior, cuando se llama al readFile Node.js le cede la operación al sistema operativo por lo que permite al programa seguir trabajando sin bloquearse. La operación I/O cuando termina lanza un evento para que Node.js lo atienda. Dicho evento en nuestro caso ejecutaría `miCallback`.
Este uso de eventos proviene de Javascript cuya ejecución se basa en lo que se llama event loop (bucle de eventos). El event loop es una cola donde se van añadiendo los bloques de código que quieren ejecutarse al finalizar una operación asincrónica.
Para verlo gráficamente:



Ejercicios

Preparación de entorno

Sistema operativo utilizado: Linux Mint 17 (basado en Ubuntu).

Instalar Node.js

Ejecutar en una terminal:

```
$ sudo apt-get install nodejs
```

Instalar npm

Ejecutar en una terminal:

```
$ sudo apt-get install npm
```

Instalar todos los módulos/dependencias que se utilizan en los ejemplos

Ubicarse en el directorio principal de los ejemplos donde se encuentra el archivo **package.json**
En este archivo, se encuentra reflejada la configuración del proyecto de Node.js tales como:

- Nombre del proyecto.
- Autor.
- Versión.
- Dependencias.
- Scripts.
- Repositorio Git.
- Motor de Node.js.

Para instalar los módulos ejecutar:

```
$ npm install
```

Con el comando anterior, npm por defecto instala en el directorio principal del proyecto todas las dependencias que defini en el package.json.
Una vez que terminó de ejecutarse 'npm install' deberíamos tener un nuevo directorio llamado **node_modules.**

Módulos utilizados

commander.js

Es una librería que permite crear interfaces command-line de forma simple.
La utilice para definir los comandos que se podían ejecutar en cada aplicación.
Podemos definir acciones y a cada una detallarle que opciones necesita para ejecutarse. Algo muy bueno e importante que provee es que dependiendo la definición de tus acciones, la libreria te genera automaticamente una opcion mas que es -h, --help y al ejecutarla te devuelve información de ayuda.

Código de ejemplo:

```
var program = require('commander');

program
  .option('-n, --nombre <string>', 'Opcion para insertar un nombre')
  .parse(process.argv);

if (!program.nombre) console.log('No cargaste el nombre');
```

RPC en Node.js

Módulos utilizados:

fs: Módulo del core de Node.js que permite operar con los file system.
path: Módulo del core de Node.js que brinda un conjunto de operaciones para trabajar y/o transformar paths.

dnode: An asynchronous rpc system for node.js that lets you call remote functions.

En Node.js debido a la escalabilidad que tiene para desarrollar aplicaciones orientadas a red y sumado a la flexibilidad de Javascript uno mismo puede diseñar su propio módulo RPC pero este caso voy a utilizar una implementación llamada **dnode** que ofrece un mecanismo de comunicación RPC Asincrónico.
Si partimos de un contexto purista, **dnode** se lo podría comparar con la "notación de primitivas múltiples" o SR debido a que el servidor exporta operaciones de igual forma que en RPC y por otro lado dichas operaciones son ejecutadas inicialmente por el mismo proceso por lo que tenemos Rendezvous.
Cuando digo inicialmente es porque podemos tener dentro de nuestro servidor operaciones I/O las cuales son procesadas por procesos independientes al servidor.
Por último, los clientes no son demorados cuando llaman a una operación en el servidor, por eso tenemos una comunicación PMA.

Código de ejemplo:

```
// código para crear un servidor en el puerto 5004 y que exporta una operación hello
var dnode = require('dnode');
var server = dnode({
    hello : function (name, cb) {
          // cb es una funcion callback, que es enviada como parametro desde un cliente. Se utiliza los callback para retornar un valor al finalizar la operación.
        cb('Hello '+name);
    }
});
server.listen(5004);
```

```
// código para crear un cliente que va conectarse al servidor del puerto 5004 y una vez que logró conectarse ejecuta el 'hello' exportado por el server.
var dnode = require('dnode');
var d = dnode.connect(5004);
d.on('remote', function (remote) {
    // se puede ver que el segundo parametro de 'hello' es una funcion anonima que es justamente el callback que mencionaba antes.
    remote.hello('martin', function (result) {
        console.log(result);
        d.end();
    });
});
```

Output:

```
$ node server.js &
[1] 27574
$ node client.js
Hello martin
```

Contexto

Un servidor almacena archivos para lectura/escritura y exporta 2 operaciones “read” y “write”.

Los clientes se conectan al servidor para leer o escribir sobre un archivo remoto que especifique.

Archivos

- rpc/server/server.js: Script que ejecuta un servidor en el puerto 3000 y sirve remotamente 2 funciones.
  - read: Recibe el nombre del archivo, la longitud de bytes a leer y un callback del cliente para retornar el resultado.
  - write: Recibe el nombre del archivo, el string que desea escribir y un callback del cliente para retornar el resultado de la operación.
- rpc/server/client.js: Script que permite crear un cliente y conectarse al servidor en el puerto 3000.

Caso de prueba - Read

1. Ejecutar servidor
    a. `$ node rpc/server/server.js`
2. Realizar una lectura de n bytes (ej: 45) sobre un archivo remoto (ej: “archivo1”)
    a. `$ node rpc/client/client.js read -f archivo1 -l 45`
    b. Parámetros:
        -f, --file <s> Nombre del archivo
        -l, --length <n> Longitud del buffer

3. Resultados del servidor:
        Se realizó una lectura sobre el archivo: <file_path>
4. Resultados del cliente:
        31 bytes leidos correctamente de archivo1: The big idea is “messaging”

Caso de prueba - Write

1. Ejecutar servidor
    a. `$ node rpc/server/server.js`
2. Realizar una escritura sobre un archivo remoto (ej: “archivo2”)
    a. `$ node rpc/client/client.js write -f archivo2 -d 'texto de ejemplo'`
    b. Parámetros:
        -f, --file <s> Nombre del archivo
        -d, --data <s> Datos/string a escribir en el archivo

3. Resultados del servidor:
    a. Se escribieron 16 bytes sobre el archivo: <file_path>
4. Resultados del cliente:
    a. Se escribieron 16 bytes sobre el archivo: <file_path>

Manager/Workers en Node.js

Si bien RPC aporta una gran flexibilidad en los sistemas distribuidos y permite una conexión bidireccional entre el llamador y el proceso server, nos encontramos con el

problema de que la conexión es iniciada por el cliente y una vez que recibe el resultado del servidor esta conexión se cierra.
El proceso server actúa como un gran ente que regula y procesa las peticiones de clientes.
Con esta dificultad, ¿cómo podríamos implementar un Manager/Workers donde el servidor debería poder decidir enviar mensajes a sus clientes sin esperar primero que ellos pidan algo?
Investigando me encontre que existen algunas implementaciones de **Actores** en Node.js basadas principalmente en Akka y me pareció que era excelente para codificar un ejemplo simple de Manager/Workers.

En el modelo de **actores**, cada objeto es un **actor**. Esta es una entidad que tiene una cola de mensajes o buzón y un comportamiento. Los mensajes pueden ser intercambiados entre los actores y se almacenan en el buzón. Al recibir un mensaje, el comportamiento del actor se ejecuta. El actor puede: enviar una serie de mensajes a otros actores, crear una serie de actores y asumir un nuevo comportamiento para el próximo mensaje. La importancia en este modelo es que todas las comunicaciones se llevan a cabo de forma asincrónica. Esto implica que el remitente no espera a que un mensaje sea recibido en el momento que lo envío, solo sigue su ejecución.

Una segunda característica importante es que todas las comunicaciones se producen por medio de mensajes: no hay un estado compartido entre los actores. Si un actor desea obtener información sobre el estado interno de otro actor, se tendrá que utilizar mensajes para solicitar esta información. Esto permite a los actores controlar el acceso a su estado, evitando problemas.

Módulos utilizados

net: Módulo del core de Node.js explicado al principio del documento.
usage: Usage es un simple módulo que permite obtener el consumo de memoria y procesamiento de un determinado bloque de código.
Inquirer.js: Inquirer es un módulo simple que te permite que armes una consola “más interactiva”. Uno define un conjunto de preguntas y se setea al prompt para que las muestre en la consola y uno responda.
actorify: Actorify es una librería que implementa Actores en Javascript. Internamente convierte un Duplex Stream en un Actor.

Código de ejemplo, ping-pong:

```
var net = require('net');
var actorify = require('actorify');

// creo un server utilizando el modulo net de Node.js y escuchando en
el puerto 3000
net.createServer(function(sock){
   // la creacion del server me devuelve un socket y este lo convierto
a la interfaz Actor provista por actorify
  var actor = actorify(sock);
  // funcion que se ejecuta cuando el actor server recibe un ping
  actor.on('ping', function(){
    console.log('PING');
    // envio un pong
    actor.send('pong');
  });
}).listen(3000);
// creo un client que se conecta a un server en el puerto 3000
var sock = net.connect(3000);
// connet del modulo 'net' devuelve un socket y este lo convierto a la
interfaz Actor provista por actorify
var actor = actorify(sock);

// cada un intervalo de 300ms ejecuto la funcion definida
setInterval(function(){
  // envio un ping
  actor.send('ping');
  // funcion que se ejecuta por unica vez cuando recibe un pong
  actor.once('pong', function(){
    console.log('PONG');
  });
}, 300);
```

Archivos:

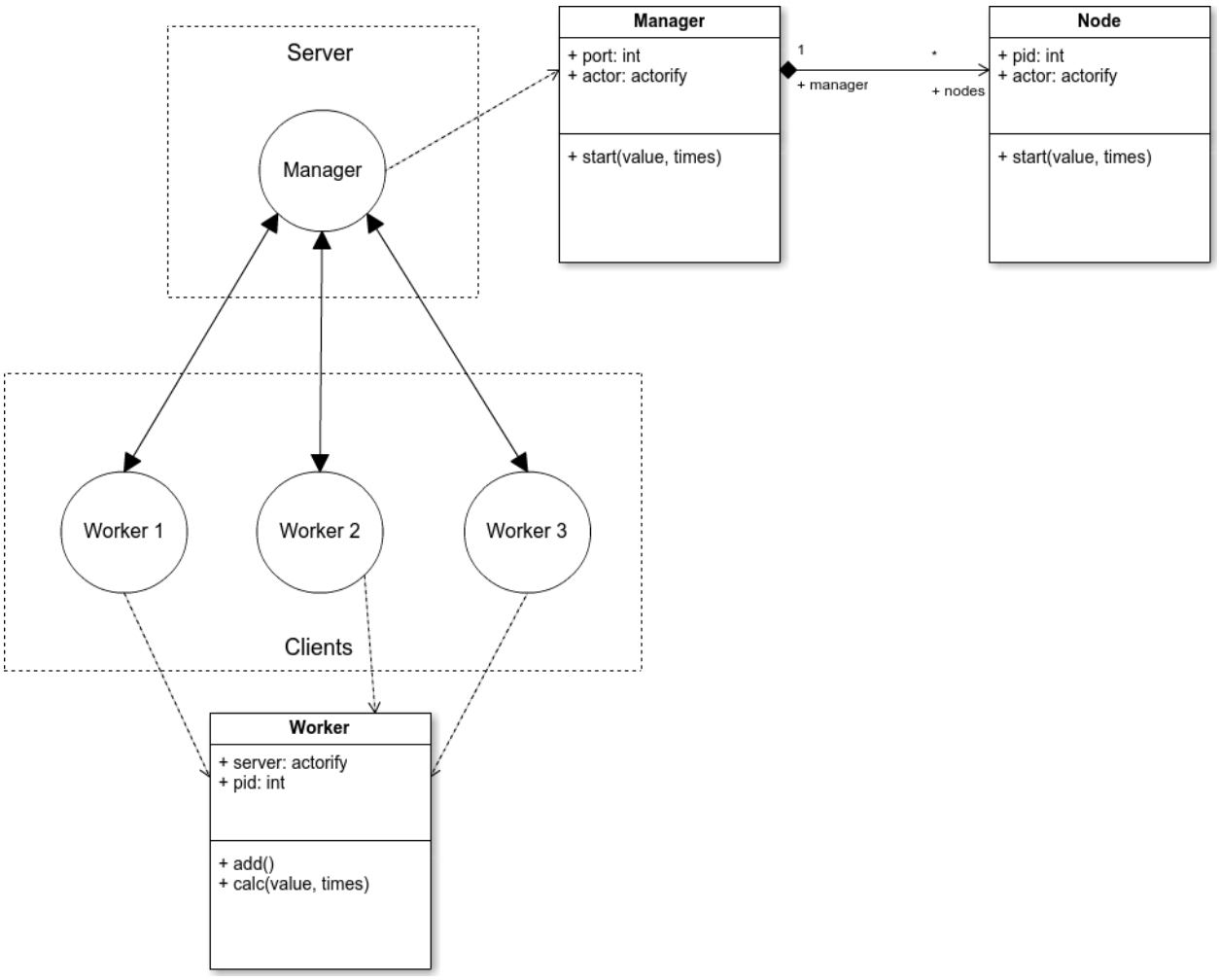
Contexto

Tenemos un servidor (manager) y uno o varios clientes (worker).

Los workers identificados por su pid se conectan al manager y esperan a que se de la orden para comenzar a ejecutar la **tarea.**
Una **tarea** es una operación que el manager delega a sus workers para que la ejecuten y retornen los resultados a el.
Para mantenerlo simple, la **tarea** es que cada worker calcule una n cantidad de veces la sucesión de fibonacci para un número determinado y auditar esa bloque de código para saber cuanta memoria y procesamiento consume. Por último, retornar el resultado de la auditoría al manager.
Cuando un worker termina su tarea (porque ya la ejecutó n veces) le notifica al manager y este le envía una señal dandole permiso a que termine por completo su ejecución.

Archivos

- mw/server.js: Es un script que solo provee la lógica para interactuar mediante la consola y así ejecutar el objeto Manager que vendría a ser nuestro servidor real.
- mw/manager.js: Es un módulo que implemente para llevar la funcionalidad del Manager a una orientación de objetos. De esa forma se me hacia mas fácil desarrollar su lógica y tener una mejor separación de roles.
    - Existe un objeto Node que declaró en el Manager mismo. La razón de esa clase de objeto es que necesitaba una estructura adicional para guardar los nodos conectados a mi red.
- mw/client.js: Es un script que solo provee la lógica para interactuar mediante la consola y así instanciar un objeto Worker el cual internamente se conecta al Manager.
- mw/worker.js: De igual forma que hice con Manager también implemente una interfaz Worker con el mismo objetivo, facilitar el desarrollo.



Caso de prueba

1. Ejecutar servidor
    a. `$ node mw/server.js -p 4000 -f 5 -t 3`
    b. Parámetros:
        -p, --port <n> Puerto del servidor
        -f, --fib <n> Numero fibonacci
        -t, --times <n> Cantidad de pruebas

2. Resultado del servidor, antes de ejecutar la tarea:
    a. Servidor iniciado. Quiere comenzar las ejecución de los workers: (y/N) ?
3. Agregar workers (en distintas terminales agregar todos los worker que quiera):
    a. `$ node mw/client.js create -s 4000`
    b. Parámetros:
        -s, --server <n> Puerto del servidor

4. Resultado de cada cliente, antes de ejecutar la tarea:
    a. Worker <pid> esperando por trabajo.
5. Iniciar la ejecución de los workers desde la terminal del servidor: y
6. Resultados del servidor teniendo 2 workers (22098 y 22128):

    Resultado del worker 22098:
    { memory: 13172736, cpu: 0.07158196134574236 }
    Resultado del worker 22128:
    { memory: 13176832, cpu: 1.6528925619846635 }
    Resultado del worker 22098:
    { memory: 13172736, cpu: 0.08350730688935451 }
    Resultado del worker 22128:
    { memory: 13176832, cpu: 1.0657193605688795 }
    Resultado del worker 22098:
    { memory: 13172736, cpu: 0.08124576844956152 }
    Worker 22098 termino!
    Resultado del worker 22128:
    { memory: 13176832, cpu: 0.7863695937093131 }
    Worker 22128 termino!
    Termino el trabajo para todos!

7. Resultados del cliente 22098:

    Prueba 1. Calcular fibonacci para: 5
    Resultado:
    { memory: 13172736, cpu: 0.07158196134574236 }
    Prueba 2. Calcular fibonacci para: 5
    Resultado:
    { memory: 13172736, cpu: 0.08350730688935451 }
    Prueba 3. Calcular fibonacci para: 5
    Resultado:
    { memory: 13172736, cpu: 0.08124576844956152 }
    Termine!

8. Resultados del cliente 22128:

    Worker 22128 esperando por trabajo.
    Prueba 1. Calcular fibonacci para: 5
    Resultado:
    { memory: 13176832, cpu: 1.6528925619846635 }
    Prueba 2. Calcular fibonacci para: 5

Resultado:
{ memory: 13176832, cpu: 1.0657193605688795 }
Prueba 3. Calcular fibonacci para: 5
Resultado:
{ memory: 13176832, cpu: 0.7863695937093131 }
Termine!

Conclusión

Como mencionaba anteriormente la diferencia de diseñar RPC en Node.js es su capacidad asincrónica en la llamada a un mensaje. Esto se debe a un concepto que está fuertemente relacionado con la forma en que Javascript maneja la concurrencia. Entendemos en primer lugar que la máquina virtual de javascript (en el caso de Node.js, la v8) corre sobre un único thread (característica popular en los lenguajes interpretados). **Pero** eso no quiere decir que no podamos tener concurrencia en Node.js.
Javascript maneja las operaciones I/O de forma concurrente y son atendidas mediante eventos.

```
call('operacion I/O').whenFinish(function() {
       //evento que se ejecuta al finalizar la operación I/O
});
```

Es por esta razón que se recomienda siempre que Node.js se utilice para esquemas con gran procesamiento de I/O.
Si consideramos que cualquier operación que abre un socket y utiliza la red es una operación de I/O, entendemos la razón de que RPC en Node.js sea sincrónico.

Después de ver como podíamos implementar RPC en Node.js, vimos las capacidades que tiene Node.js para portar librerías orientadas al desarrollo de aplicaciones distribuidas como el caso de Akka y sus Actores.
Mediante los **Actores** logre extender el nodo servidor para que envíe mensajes a todos los nodos clientes conectados en la red. De esta forma podemos resolver infinidad de de problemas distribuidos.