Introducción a

NODE.JS v0.8

a través de

koans

Arturo Muñoz de la Torre

INTRODUCCIÓN A NODE.JS A TRAVÉS DE KOANS

Arturo Muñoz de la Torre Monzón (@arturomtm)
Ingeniero de Teleco por la Universidad Politécnica de Madrid. Entusiasta de las tecnologías que dan vida a la Web. Curioso e imaginativo, siempre en evolución.

Introducción a Node.JS a través de Koans

por Arturo Muñoz de la Torre Monzón

http://nodejskoans.com

Revisión del texto:
D. Carlos Ángel Iglesias Fernández
Profesor en la Escuela Técnica Superior de Ingenieros de
Telecomunicación de la Universidad Politécnica de Madrid.

El diseño de la portada ha corrido a cargo del autor del libro, empleando para ello las tipografías: Bebas Neue, una versión modificada de la Aharoni Bold, Comfortaa, Tribal Animals Tatto Design (http://tattoowoo.com/) y Entypo pictograms by Daniel Bruce (www.entypo.com).

2013-05-23

A mis padres, Arturo y Fabi, y a mi hermano Edu, por su apoyo y confianza sin límites.

A Irene, porque sin ella no hubiera llegado nunca al final.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

Con el presente proyecto se pretenden establecer unas líneas de aprendizaje de la plataforma Node a través de un recorrido interactivo por sus principales módulos, tanto propios como de terceras partes, con el que introducirse y asentar sus principios básicos de funcionamiento. A través de seis capítulos se quiere proporcionar al lector de una amplia visión de qué herramientas pone Node a su disposición y cómo debe usarlas para realizar sus propios desarrollos.

Cada uno de estos capítulos tratará uno o dos módulos muy relacionados entre sí, con la intención de que se cubran tópicos que vayan de menor complejidad a mayor. La estructura de dichos capítulos es siempre la misma:

- un ligero repaso a los conceptos teóricos que hay detrás del módulo, generalmente desgranando una RFC (*Request For Comments*), con especial énfasis en aquellos que el API da opción a manejar.
- un recorrido por el API del módulo no sintetizado, como pueda aparecer en la documentación de Node, sino explicando la relaciones intrínsecas entre los métodos, propiedades y eventos del módulo.
- una propuesta de aplicación a desarrollar que esté basada en el módulo que se está tratando. Se procurará que la aplicación cubra la mayor parte posible de los conceptos que se destacaron en la introducción teórica.
- el diseño propuesto al anterior problema, focalizando el desarrollo en aquellos puntos que usan las herramientas que el módulo de Node proporciona para cumplir con los requisitos que la especificación anterior del problema

impone.

- una selección de fragmentos de código, cuyo dominio se considera obligatorio para el conocimiento mínimo necesario del módulo, de tal manera que permitan desarrollar una aplicación con él. Estos fragmentos son los candidatos para convertirse en Koans. Los Koans son la técnica de aprendizaje escogida para el proyecto. En los los párrafos siguientes se describirá qué son.

La selección de módulos que se ha realizado viene motivada esencialmente por un factor: Node está pensado para aplicaciones de red. Tiene sentido, por tanto, que sean todos aquellos relacionados con este hecho. Y no sólo dentro de los módulos nativos de la plataforma, sino también aquellos que su popularidad entre la comunidad de desarrolladores los ha hecho un estándar de facto. El proyecto comprenderá pues los siguientes capítulos:

- Capítulo 3: módulos *dgram* y *Buffer*, o UDP como primera parada para ir cogiendo la dinámica de la plataforma.
- Capítulo 4: módulos *net* y *stream*, para introducir con TCP un punto de complejidad en los protocolos de transporte.
- Capítulo 5: módulo *http*, con el que subir un nivel en la pila de protocolos.
- Capítulo 6: módulos *Express*, como la herramienta perfecta para la creación de aplicaciones web.
- Capítulo 7: módulo *Socket.IO*, el paradigma de protocolo para las aplicaciones en tiempo real.

Como se ha comentado unas líneas atrás, el método elegido para el aprendizaje interactivo de la plataforma son los Koans, pero...¿qué son los Koans?

1.1. ¿Qué son los Koans?

El término *koan* proviene de la filosofía Zen oriental. Es el nombre que se le da a un pequeño problema que un maestro plantea a su discípulo para evaluar su progreso. Suelen ser breves y escuetos y, a menudo, con apariencia trivial o absurda pero que, sin embargo, suponen para el discípulo un paso más en su

camino hacia “el despertar” puesto que suponen para él realizar un esfuerzo para entender la doctrina.

Los koans en el ámbito informático fueron idea de Joe O’Brien y Jim Weirich, de EdgeCase, como forma de alcanzar “la iluminación” en el aprendizaje del lenguaje Ruby. Manteniendo la analogía con la filosofía Zen, los koans de Ruby son pequeños trozos de código donde se sustituye alguna parte con unos guiones bajos ‘__’, al estilo de los ejercicios de *rellenar el hueco*, que el alumno deberá cambiar por el código que piense que es el correcto para que el ejercicio ejecute sin errores. Para evaluar si lo que el alumno ha escrito en lugar de los ‘__’ es correcto se emplean unos casos de prueba unitarios diseñados para cada koan.

Posteriormente los koans se han extendido a más lenguajes de programación, entre los que se incluyen JavaScript o Python.

La filosofía de los Koans que se han diseñado para Node es la misma que la que se ha presentado para los lenguajes de programación: fragmentos de código evaluables mediante pruebas unitarias que marquen al alumno el camino a seguir para la correcta comprensión y asimilación de los aspectos relevantes necesarios para iniciarse en Node. Estos trozos de código se engloban en el marco de una aplicación concreta, con lo que no son independientes entre sí, sino que su resolución lleva al alumno a tener en sus manos un programa con completa funcionalidad donde entender mejor cuáles son los puntos clave más básicos cuando se programa con Node como tecnología de Servidor.

1.2. Guía de lectura

Con la intención con la que se crearon los Koans se crea esta obra, cuyos capítulos van marcando el camino hacia el conocimiento y manejo de Node. El recorrido que se marca para ello es el que sigue:

Capítulo 2

El punto de partida que se va a tomar para introducir la programación de aplicaciones para Internet es uno de sus protocolos más sencillos: UDP. Con él se presentará el módulo *dgram*, que implementa las características de este protocolo, y *Buffer*, íntimente ligado a los protocolos de red porque es la manera que tiene Node de manejar datos binarios sin un formato concreto, en bruto.

Con *dgram* se tomará contacto con la creación y gestión de conexiones UDP y, sobre todo, con todo lo referente a operaciones de Entrada/Salida asíncrona en redes, como envío y recepción de datos.

Con *Buffer*, se conocerá una de las clases básicas de Node, siempre presente en el espacio de ejecución de cualquier programa, por lo que resulta imprescindible conocer cómo se crean y se manejan los datos que contiene a través de los múltiples métodos que *Buffer* ofrece para ello.

El pegamento que junta ambos módulos es una aplicación que implementa el protocolo RTP (*Real Time Protocol*), basado en UDP, para transmitir canciones a los clientes que lo soliciten.

Capítulo 3

Siguiendo con los protocolos de red, se da un paso más con respecto al tema anterior y se introduce el módulo *net*, que da soporte a la transmisión de datos sobre el protocolo TCP, "hemano mayor" de UDP. Estos datos se ven como un flujo continuo, o *stream*, con lo que la inclusión del módulo *stream* en este capítulo cobra sentido.

TCP introduce mayor complejidad, lo que se traduce en mayor número de parámetros a controlar en una transmisión, completando así los conocimientos adquiridos en el tema anterior sobre comunicación Cliente-Servidor. Ahora se introducirán los métodos específicos para TCP que se usan para gestionar el ciclo de vida de una conexión: desde su generación a su cierre, pasando por la lectura y escritura de datos en ella.

El paso que se avanza con este capítulo se refleja en la aplicación ejemplo que se plantea como problema: es una evolución de la anterior, donde ahora se incorpora soporte TCP para el control del envío del audio a los clientes. Se conseguirá introduciendo un servidor TCP que acepte conexiones por las que se reciban comandos en modo texto.

Capítulo 4

Por encima del nivel de transporte, en el que se ubicarían los protocolos vistos en los capítulos anteriores, se encuentra el nivel de aplicación para el que Node también posee módulos en su API. En concreto, para el Protocolo de Transferencia de Hipertexto, HTTP, en el que se basa toda la Web.

La naturaleza de este módulo es similiar a la de los anteriores puesto que

sigue el modelo Cliente-Servidor ya asimilado, pero se añaden más características propias de la especificación HTTP. Por ello, junto al manejo de conexiones HTTP se introducirán los mecanismos para gestionar peticiones interpretando los métodos y cabeceras propias del protocolo para generar las respuestas de la manera más adecuada.

Aunque HTTP se sitúe por encima de TCP, no implica que no pueda convivir con más protocolos en una misma aplicación. En este caso, en la aplicación de ejemplo, se utilizará junto con RTP para proporcionar una interfaz web con la que controlar qué se emite por *streaming*, simulando un reproductor de audio tradicional.

Capítulo 5

Una vez que se conocen las librerías fundamentales de Node es hora de evolucionar y subir un peldaño en la arquitectura de los módulos de terceras partes para la plataforma. Visto HTTP, se presentará la manera de tratar con él desde otra perspectiva: la de la creación de Aplicaciones Web.

Si se sigue el movimiento en la pila de una aplicación web para Node puede encontrarse con Connect y Express, dos de las librerías más famosas del entorno. Ambas hacen uso de los módulos que ofrece Node para poner a disposición del programador un *middleware* (Connect) y un framework web (Express). A lo largo del capítulo se aprenderá a configurar Connect para incorporar las diversas funcionalidades que ofrece y a hacer uso de la funcionalidad enriquecida que añade Express a las peticiones y respuestas HTTP, así como a entender el nivel de abstracción que supone respecto del módulo nativo *http* de Node.

Junto con ellas, se hablará de Mongoose, un *driver* para Node para la base de datos no relacional MongoDB. Este tipo de bases de datos están muy presentes en el desarrollo de las aplicaciones web más modernas y, como toda base de datos, son imprescindibles a la hora de desarrollar una solución completa. Por ser un *driver* muy completo y complejo, se presentarán las operaciones más elementales que se pueden realizar con él para emprender un desarrollo básico pero perfectamente funcional.

Capítulo 6

El último peldaño antes de adquirir una base más o menos amplia sobre la programación para Internet con Node, son las aplicaciones en tiempo real,

las cuales están adquiriendo, si es que no lo han hecho ya, una importancia fundamental en cualquier desarrollo web.

Con este capítulo se aprenderán algunos de los conceptos realacionados con ellas, y se afianzarán con un módulo, Socket.IO, diseñado para hacerlo de una manera eficiente y sencilla. Se presentará su funcionamiento, cómo realiza y gestiona todas las conexiones, tanto en la parte servidor como en la cliente, y cómo se produce el intercambio de datos entre los mismos.

Siguiendo el mismo esquema que en capítulos anteriores, se concluirá con el desarrollo de una aplicación que haga uso de las mínimas partes necesarias para obtener un comportamiento que responda a las características de tiempo real. Para apreciarlas, qué mejor que un juego donde una interacción instantánea es obligada para una correcta experiencia de usuario.

1.3. Conseguir el código fuente

Todo el código que se ha generado para este proyecto puede localizarse en Github[1], una comunidad de desarrolladores. El repositorio concreto donde se hallan todos los archivos es uno en la cuenta personal del autor del libro, al que se ha llamado `nodejskoans`: https://github.com/arturomtm/nodejskoans.git. Para tener una copia con la que trabajar, es necesario clonarlo en un directorio de la máquina de trabajo:

```
$ mkdir nodejskoans
$ cd nodejskoans
$ git clone https://github.com/arturomtm/nodejskoans.git
```

Una vez clonado, se puede trabajar con el código tal y como se explica en los capítulos que siguen.

[1] http://github.com

Capítulo 2

Introducción a Node v0.8

Node.js, de ahora en adelante *Node*, es un proyecto creado por Ryan Dahl a principios de 2009. Se diseñó orientado a la creación de aplicaciones para Internet, principalmente Web, porque la programación de software para servidores era el tipo de desarrollo que hacía el autor en aquella fecha.

La idea empezó a gestarse a partir de otro proyecto para el *framework Ruby on Rails*, un pequeño y rápido servidor web llamado Ebb, también de la autoría de Ryan Dahl, que evolucionó a una librería en C [3]. El aspecto negativo de esto era la complejidad que supone el lenguaje C para programar aplicaciones basadas en dicha librería. Es en este punto donde entra en escena el, por aquel entonces, recién aparecido intérprete de JavaScript de Google, V8, que no tardó en adoptarse como el motor de la plataforma emergente.

Una de las razones de la evolución del proyecto desde Ruby a C, y luego de C a JavaScript fue el objetivo de realizar un sistema en que la Entrada/Salida fuera enteramente no bloqueante. Ryan tenía claro que era esencial para obtener un alto rendimiento. Con Ruby y C siempre había una parte del sistema que era bloqueante. Pero JavaScript se ajusta a este requisito porque está diseñado para ejecutarse en un bucle de eventos, que es, precisamente, lo que Node hace: delegar en la plataforma las operaciones de Entrada/Salida que solicita la aplicación. De esta manera Node puede seguir realizando tareas sin estar bloqueado esperando, y cuando las operaciones se hayan completado, procesará en su bucle de eventos el evento generado y ejecutará el código que lo maneja según se haya definido. La consecuencia de este modelo Entrada/Salida es que se puede atender a un altísimo número de clientes a la vez, motivo que ha llevado a Node a ser el

paradigma de plataforma de aplicaciones de tiempo real.

Desde la presentación de Node, el proyecto no ha parado de crecer. Actualmente, es el segundo repositorio más popular en Github[1], con más de 20.000 seguidores, y tiene más de 24.794 librerías, a las que llaman módulos, registradas en la web de su gestor de paquetes, NPM[2]. Todo ello estando sólo, a fecha de redacción de este proyecto, en la versión 0.8 (rama estable).

Además de la gran actividad en Internet, a la que hay que sumarle el concurso mundial de aplicaciones Node Knockout[3], la comunidad de Node tiene una cita anualmente con el ciclo de conferencias NodeConf[4], donde se presentan todos los avances de la plataforma, o con la JSConf[5], un evento para desarrolladores de JavaScript donde la plataforma es protagonista de muchas de las conferencias que se realizan.

Node está apadrinado por la compañía Joyent[6], que contrató a Ryan Dahl cuando comenzaba el proyecto. Joyent ofrece, conjuntamente con Nodejistsu[7], como *IaaS*, un entorno en "la Nube" donde desplegar aplicaciones Node.

Pero no sólo Joyent ofrece alojamiento para esta plataforma. Heroku[8] también tiene soluciones *cloud-computing* personalizables, y, si se eligen opciones gratuitas y *open source*, Nodester[9] da espacio para aplicaciones como *PasS*.

Como colofón, Node fue galardonado en 2012 con el premio "Tecnología del Año" por la revista InfoWorld, perteneciente a una división prestigioso grupo internacional de prensa especializada IDG. Y, posiblemente, todo no haya hecho más que empezar.

[1] https://github.com
[2] https://npmjs.org/
[3] http://nodeknockout.com/
[4] http://www.nodeconf.com
[5] http://jsconf.com
[6] http://joyent.com
[7] https://www.nodejitsu.com/
[8] http://www.heroku.com
[9] http://nodester.com

2.1. ¿Qué es Node?

La mejor manera de aproximarse a Node es a través de la definición que aparece en su página web: [4]

"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 ligero y eficiente, perfecto para aplicaciones data-intensive en tiempo real"

Esta visión global de Node se puede diseccionar en pequeñas partes que, una vez analizadas separadamente, dan una visión mucho más precisa y detallada de las características del proyecto.

2.2. "Es una plataforma"

En efecto, Node 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, aunque una parte de estas librerías permite interactuar con componentes del sistema operativo a través de funciones que cumplen con el estándar POSIX.

Básicamente este estándar o familia de estándares define las interfaces y el entorno, así como utilidades comunes, que un sistema operativo debe soportar y hacer disponibles para que el código fuente de un programa sea portable (compilable y ejecutable) en diferentes sistemas operativos que implementen dicho estándar. En este caso, Node facilita funciones para manejo de archivos, Entrada/Salida, señales y procesos conformes a las características establecidas por POSIX [5].

2.2.1. El proceso de arranque de Node

El código que realiza el arranque, o *bootstrap*, del núcleo de la plataforma es lo primero que se ejecuta y uno de sus cometidos es proveer el mecanismo para cargar el resto de los módulos del core según vayan siendo necesarios o se

demanden. Se puede echar un vistazo a este código en el fichero del código fuente `src/node.js`. Ahí se ve que el encargado de realizar la carga es el objeto `NativeModule`, que ofrece un minimalista sistema de gestión de módulos.

Pero aparte de `NativeModule`, en el proceso de arranque ocurren muchas más cosas, de las que se encargan el resto de funciones presentes en `node.js`.

Una de las primeras acciones que se realizan es hacer disponibles las variables, objetos y funciones globales. Por globales se entiende que están disponibles en el *scope* donde corre Node, que se hace disponible al programa mediante, precisamente, la variable `global`. Por defecto, disponemos de las funciones que ofrecen los módulos *console* y *timers*, el objeto `Buffer`, del módulo *buffer*, y el objeto nativo `process`.

Se hablará de `process` por ser quizás uno de los objetos más interesantes de la plataforma desde el punto de vista del diseño. Presente en el espacio global de ejecución del proceso principal de Node representa a ese mismo proceso. Como se ha dicho, se hace disponible en él después de crearse en código nativo a través de la función `SetupProcessObject()` en el proceso de arranque definido en el fichero `src/node.cc`[10]. Con esta función se crea un objeto al que se le añaden propiedades y métodos en código nativo que luego están disponibles para el programador a través del API. Éstos permiten:

- identificar la arquitectura y sistema operativo donde corre Node, mediante las propiedades `process.arch` y `process.platform`, o con `process.features`
- tener conocimiento mínimo sobre el proceso de Node como su identificador de proceso con `process.pid`, el directorio de trabajo con `process.cwd()`, que se puede cambiar con `process.chdir()`, el path desde donde se inició con `process.execPath`, o las variables de entorno del usuario con `process.env`
- conocer versiones de Node y de librerías nativas con las propiedades `process.version` y `process.versions`
- en caso de sistemas *nix, manejar identificadores de grupos y usuarios con `process.getuid()`, `process.setuid()`, `process.getgid()` y `process.setgid()`

[10] node.cc https://github.com/joyent/node/blob/v0.8.20-release/src/node.cc#L2166

- estadísticas de ejecución como el tiempo que lleva corriendo el proceso, con `process.uptime()` y la memoria que está consumiendo con `process.memoryUsage()`

Además, existen una serie de variables y métodos no documentados del todo que son interesantes desde el punto de vista de funcionamiento interno de Node:

- se tiene un registro de los módulos que se han cargado en la plataforma, en el array `process.moduleLoadList`. En esta lista identificaremos dos tipos de módulos por la etiqueta que precede al nombre del módulo: `NativeModule` y `Binding`.

  `NativeModule` se refiere a los módulos JavaScript que componen el core de Node y se cargan a través del objeto `NativeModule`.

  `Binding`, por su parte, identifica a los módulos escritos en C/C++ de más bajo nivel que los otros. Estos *bindings* o *addons*, ofrecen sus métodos o sus objetos al código de un programa como si de otro módulo más se tratara, o, en la arquitectura de Node, sirven como base a las librerías en JavaScript. Un *binding* se carga internamente con el método `process.binding()`, que no debería ser accesible para el desarrollador, aunque lo es y se puede invocar.

  Reseñar que `process.binding()` no tiene nada que ver con `process.dlopen()`, otro método que también se puede invocar, aunque debe hacerlo siempre el cargador de módulos, y que sirve para cargar los *addons* de terceras partes o que no forman parte del núcleo de Node. Son dos maneras diferentes de cargar librerías, que en el fondo hacen lo mismo pero con distinta finalidad.

- se tiene también estadísticas de diagnóstico del mencionado bucle de eventos a través de `process.uvCounters()`. Obtendremos los contadores internos del bucle de eventos que se incrementan cada vez que se registra un evento en él.

Seguidamente, se termina de inicializar el objeto `process` ya a nivel de código no nativo, sino JavaScript:

- Se establece la clase `EventEmitter` su como prototipo, con lo cual `proccess` hereda sus métodos y por tanto la capacidad de generar eventos y notificarlo a sus subscriptores. Se profundizará en `EventEmitter` cuando se hable de

la arquitectura de Node.

- Se establecen los métodos de interacción con el bucle de eventos [11]. El bucle de eventos es una característica de la arquitectura de Node que tiene mucho que ver con el altísimo rendimiento de la plataforma. Más adelante se analizará en detalle este punto. A nivel de programador, el API ofrece `process.nextTick()`.

- Se inicializan los *Streams* de Entrada/Salida estándar [12]. El API los presenta como las propiedades: `process.stdout`, `process.stderr`, habilitados ambos para la escritura y, a diferencia de los demás *Streams* de Node, son bloqueantes; y `process.stdin` que es de lectura y está pausado de inicio cuando la entrada estándar no es otro *Stream*, con lo que, para leer de él, se debe abrir primero invocando a `process.openStdin()`.

- Se definen los métodos relacionados con las señales[13] del sistema operativo: `process.exit()`, para terminar la ejecución del programa especificando un código de salida, y `process.kill()`, para enviar señales a otros procesos, tal y como se haría con la llamada al sistema operativo *kill*.

- Se añade funcionalidad a los métodos para manejar listeners[14] que hereda de EventEmitter, recubriéndolos para que sean capaces de escuchar y actuar sobre señales del sistema operativo (tipo Unix), no sólo eventos.

- Se determina si, en lugar de ser el proceso principal de Node, el programa es un proceso *worker* del modulo *cluster*, de los que se hablará cuando se comente el entorno de ejecución de Node.

Por último, se determinar en qué modo se va a ejecutar Node y se entra en él. Actualmente hay varios modos:

Script

se accede a este modo pasando por línea de comandos el nombre de un fichero `.js` que contiene el código del programa. Node cargará ese fichero y lo ejecutará. Es el modo más común de trabajar en la plataforma.

También se puede indicar por línea de comandos mediante el modificador *-e*

[11] `node.js` https://github.com/joyent/node/blob/v0.8.20-release/src/node.js#L47
[12] `node.js` https://github.com/joyent/node/blob/v0.8.20-release/src/node.js#L352
[13] `node.js` https://github.com/joyent/node/blob/v0.8.20-release/src/node.js#L432
[14] `node.js` https://github.com/joyent/node/blob/v0.8.20-release/src/node.js#L464

o *–eval* seguido del nombre del script.

REPL
: son las siglas de *Read Eval Print Loop*. Es el modo interactivo en el que Node presenta un prompt ('>') donde manualmente se van introduciendo expresiones y comandos que serán evaluados al presionar la tecla Enter.

Debug
: se invoca incluyendo el modificador debug en la línea de comandos y deja a Node en un modo interactivo de depuración de scripts donde se pueden introducir mediante comandos las acciones típicas en este tipo de sesiones: step in, step out, fijar y limpiar breakpoints...

Adicionalmente, los autores del proyecto permiten cargar código propio del programador y arrancar con él en lugar de hacerlo con el arranque normal de Node. Para ello se debe incluir en el directorio `lib/` del código fuente del proyecto el fichero `_third_party_main.js` que contendrá el código personalizado, y luego compilarlo todo.

El objeto que carga las librerías, como se ha comentado, es `NativeModule` el cual ofrece un pequeño mecanismo para la carga y gestión de módulos. Este mecanismo es auxiliar y una vez cumple su función se reemplaza por la funcionalidad, más completa, que ofrece el módulo *module*, cargado a su vez, paradójicamente, por `NativeModule`. Cuando se desea tener disponible cualquier módulo para usarlo en un programa, el método `require()` que se invoca es el de `module`[15]. Este método comprueba, con ayuda de NativeModule, si el módulo está en la caché de módulos y en caso negativo, lo busca en el sistema de ficheros o en caso de que sea un módulo del core de Node, en el propio ejecutable de la plataforma, ya que éstos están embebidos en él [6]. Una vez localizado, lo compila, lo mete en la caché y devuelve su `exports`, es decir, las funciones, variables u objetos que el módulo pone a disposición del programador.

Las librerías del core son también archivos de extensión `.js` que se hallan en el directorio `lib/` del código fuente (pero empotrados en el ejecutable una vez compilado, como se ha dicho). Cada uno de esos archivos es un módulo que sigue el formato que define *CommonJS*.

[15] `module.js` https://github.com/joyent/node/blob/v0.8.20-release/lib/module.js#L377

2.2.2. El formato CommonJS

CommonJS es una creciente colección de estándares que surgen de la necesidad de completar aspectos que se echan en falta en la especificación de JavaScript, el lenguaje de programación usado en Node. Entre estos aspectos perdidos se pueden nombrar la falta de una API estándar y bien definida, la de unas interfaces estándar para servidores web o bases de datos, la ausencia de un sistema de gestión de paquetes y dependencias o, la que de mayor incumbencia: la falta de un sistema de módulos.[7]

Node soporta e implementa el sistema que *CommonJS* define para esta gestión de módulos, lo cual es importante no sólo a la hora de organizar el código sino a la hora de asegurar que se ejecuta sin interferir en el código de los demás módulos aislándolos unos de otros de tal manera que no haya conflicto entre, por ejemplo, funciones o variables con el mismo nombre. A esto se le conoce como *scope isolation*.

Las implicaciones del uno de este estándar son:[8]

- El uso de la función `require()` para indicar que queremos emplear una determinada librería pasándole el identificador de la librería (su nombre) como parámetro.
- La existencia de la variable `exports` dentro de los módulos. Esta variable es un objeto que es el único modo posible que tiene un módulo de hacer públicas funciones y demás objetos, añadiéndolas a exports conforme se ejecuta el código de dicho módulo.
- La definición dentro de un módulo de la variable `module`. Esta variable es un objeto con la propiedad obligatoria `id` que identifica unívocamente al módulo y por tanto, se obtiene de él el exports que interese al desarrollador.

En definitiva, se puede hacer uso de las características que ofrezca un módulo, siempre y cuando éste las tenga añadidas a su `exports`, si se indica en el código con `require('modulo')`.

Por ejemplo, el siguiente módulo, al que se llamará 'circle' y se usará en un programa código gracias a `require('circle')`, permite calcular el área y perímetro de cualquier circulo; sin embargo, no permite conocer el valor de la constante PI ya que no la incluye en su `exports`:

```
var PI = 3.14;
exports.area = function (r) {
  return PI * r * r;
};
exports.circumference = function (r) {
  return 2 * PI * r;
};
```

2.2.3. Módulos disponibles en el core de Node

Se dispone pues de una serie de módulos que conforman el núcleo de Node y que se pueden usar en las aplicaciones. Para utilizar la gran mayoría de estas librerías se debe indicar explícitamente, mediante la función `require()`, que, como se ha visto, es el mecanismo que Node ofrece para tener disponibles los `exports` de los módulos. Sin embargo, como se ha comentado antes, hay una serie de módulos que están disponibles implícitamente, ya que se cargan en el proceso de bootstrap presente en `src/node.js`. El API los denomina `Globals` y ofrecen distintos objetos y funciones accesibles desde todos los módulos, aunque algunos de ellos sólo en el ámbito (*scope*) del módulo, no en el ámbito global (de programa). Estos módulos son:

console
: marcado en el API como STDIO, ofrece el objeto `console` para imprimir mensajes por la salida estándar: *stdout* y *stderr*. Los mensajes van desde los habituales `info` o `log` hasta trazar la pila de errores con `trace`.

timers
: ofrece las funciones globales para el manejo de contadores que realizarán la acción especificada pasado el tiempo que se les programa. Debido a la cómo está diseñado Node, relacionado con el bucle de eventos del que se hablará en un futuro, no se puede garantizar que el tiempo de ejecución de dicha acción sea exactamente el marcado, sino uno aproximado cercano a él, cuando el bucle esté en disposición de hacerlo.

module
: proporciona el sistema de módulos según impone *CommonJS*. Cada módulo que se carga o el propio programa, está modelado según *module*, que se verá como una variable, `module`, dentro del mismo módulo. Con ella se tienen dis-

ponibles tanto el mecanismo de carga `require()` como aquellas funciones y variables que exporta, en `module.exports`, que destacan entre otras menos corrientes que están a un nivel informativo: módulo que ha cargado el actual (`module.parent`), módulos que carga el actual (`module.children`)...

buffer

es el objeto por defecto en Node para el manejo de datos binarios. Sin embargo, la introducción en JavaScript de los *typedArrays* desplazará a los *Buffers* como manera de tratar esta clase de datos [9].

Los módulos siguientes, listados por su identificador, también forman parte del núcleo de Node, aunque no se cargan al inicio, pero se exponen a través del API:

util

conjunto de utilidades principalmente para saber de si un objeto es de tipo array, error, fecha, expresión regular...También ofrece un mecanismo para extender clases de JavaScript a través de herencia:

```
inherits(constructor, superConstructor);
```

events

provee la fundamental clase `EventEmitter` de la que cualquier objeto que emite eventos en Node hereda. Si alguna clase del código de un programa debe emitir eventos, ésta tiene que heredar de `EventEmitter`.

stream

interfaz abstracta que representa los flujos de caracteres de Unix de la cual muchas clases en Node heredan.

crypto

algoritmos y capacidades de cifrado para otros módulos y para el código de programa en general.

tls

comunicaciones cifradas en la capa de transporte con el protocolo TLS/SSL, que proporciona infraestructura de clave pública/privada.

string_decoder

proporciona una manera de, a partir de un *Buffer*, obtener cadenas de ca-

racteres codificados en *utf-8*.

fs

funciones para trabajar con el sistema de ficheros de la manera que establece el estándar POSIX. Todos los métodos permiten trabajar de forma asíncrona (el programa sigue su curso y Node avisa cuando ha terminado la operación con el fichero) o síncrona (la ejecución del programa se detiene hasta que se haya completado la operación con el fichero).

path

operaciones de manejo y transformación de la ruta de archivos y directorios, a nivel de nombre, sin consultar el sistema de ficheros.

net

creación y manejo asíncrono de servidores y clientes, que implementan la interfaz *Stream* mencionada antes, sobre el protocolo de transporte TCP.

dgram

creación y manejo asíncrono de datagramas sobre el protocolo transporte UDP.

dns

métodos para tratar con el protocolo DNS para la resolución de nombres de dominio de Internet.

http

interfaz de bajo nivel, ya que sólo maneja los *Streams* y el paso de mensajes, para la creación y uso de conexiones bajo el protocolo HTTP, tanto del lado del cliente como del servidor. Diseñada para dar soporte hasta a las características más complejas del protocolo como *chunk-encoding*.

https

versión del protocolo HTTP sobre conexiones seguras TLS/SSL.

url

formateo y análisis de los campos de las URL.

querystrings

utilidades para trabajar con las *queries* en el protocolo HTTP. Una *query* son los parámetros que se envían al servidor en las peticiones HTTP. Dependiendo del tipo de petición (`GET` o `POST`), pueden formar parte de la URL por lo

que deben codificarse o escaparse y concatenarse de una manera especial para que sean interpretadas como tal.

readline
: permite la lectura línea por línea de un *Stream*, especialmente indicado para el de la entrada estándar (*STDIN*).

repl
: bucle de lectura y evaluación de la entrada estándar, para incluir en programas que necesiten uno. Es exactamente el mismo módulo que usa Node cuando se inicia sin argumentos, en el modo *REPL* comentado con anterioridad.

vm
: compilación y ejecución bajo demanda de código.

child_process
: creación de procesos hijos y comunicación y manejo de su entrada, salida y error estándar con ellos de una manera no bloqueante.

assert
: funciones para la escritura de tests unitarios.

tty
: permite ajustar el modo de trabajo de la entrada estándar si ésta es un terminal.

zlib
: compresión/descompresión de *Streams* con los algoritmos *zlib* y *gzip*. Estos formatos se usan, por ejemplo, en el protocolo HTTP para comprimir los datos provenientes del servidor. Es conveniente tener en cuenta que los procesos de compresión y descompresión pueden ser muy costosos en términos de memoria y consumo de CPU.

os
: acceso a información relativa al sistema operativo y recursos hardware sobre los que corre Node.

_debugger
: es el depurador de código que Node tiene incorporado, a través de la opción debug de la línea de comandos. En realidad es un cliente que hace uso

de las facilidades de depuración que el intérprete de Javascript que utiliza Node ofrece a través de una conexión TCP al puerto 5858. Por tanto, no es un módulo que se importe a través de `require()` sino el modo de ejecución *Debug* del que se ha hablado antes.

cluster
: creación y gestión de grupos de procesos Node trabajando en red para distribuir la carga en arquitecturas con procesadores multi-core.

punycode
: implementación del algoritmo *Punycode*, disponible a partir de la versión 0.6.2, para uso del módulo *url*. El algoritmo *Punycode* se emplea para convertir de una manera unívoca y reversible cadenas de caracteres *Unicode* a cadenas de caracteres *ascii* con caracteres compatibles en nombres de red. El propósito es que los nombres de dominio internacionalizados (en inglés, IDNA), aquellos con caracteres propios de un país, se transformen en cadenas soportadas globalmente. [10]

domain
: módulo experimental en fase de desarrollo y, por tanto, no cargado por defecto para evitar problemas, aunque los autores de la plataforma aseguran un impacto mínimo. La idea detrás de este él es la de agrupar múltiples acciones de Entrada/Salida diferentes de tal manera que se dotan de un contexto definido para manejar los errores que puedan derivarse de ellas [9]. De esta manera el contexto no se pierde e incluso el programa continua su ejecución.

Quedan una serie de librerías, que no se mencionan en la documentación del API pero que existen en el directorio `lib/` del código fuente. Estas librerías tienen propósitos auxiliares para el resto de los módulos, aunque se pueden utilizarlas a través de `require()`:

_linklist
: implementa una lista doblemente enlazada. Esta estructura de datos se emplea en `timers.js`, el módulo que provee funcionalidad de temporización. Su función es encadenar temporizadores que tengan el mismo tiempo de espera, *timeout*. Esta es una manera muy eficiente de manejar enormes cantidades de temporizadores que se activan por inactividad, como los *timeouts* de los *sockets*, en los que se reinicia el contador si se detecta actividad en

él. Cuando esto ocurre, el temporizador, que está situado en la cabeza de la lista, se pone a la cola y se recalcula el tiempo en que debe expirar el primero [11].

buffer_ieee754
: implementa la lectura y escritura de números en formato de coma flotante según el estándar IEEE754 del IEEE[16] que el módulo *buffer* emplea para las operaciones con *Doubles* y *Floats*.

constants
: todas las constantes posibles disponibles de la plataforma como, por ejemplo, las relacionadas con POSIX para señales del sistema operativo y modos de manejo de ficheros. Sólo realiza un *binding* con `node_constants.cc`.

freelist
: proporciona una sencilla estructura de *pool* o conjunto de objetos de la misma clase (de hecho, el constructor de los mismos es un argumento necesario). Su utilidad se pone de manifiesto en el módulo *http*, donde se mantiene un conjunto de *parsers* HTTP reutilizables, que se encargan de procesar las peticiones HTTP que recibe un Servidor.

sys
: es un módulo deprecado, en su lugar se debe emplear el módulo `utils`.

Todos los módulos anteriores, una vez se ha compilado la plataforma, quedan incorporados dentro del binario ejecutable, por lo que, físicamente por su nombre de archivo no son localizables en disco. Por otra parte, si en disco hubiera un módulo cuyo identificador, según *CommonJS*, coincidiera con el de algún módulo del núcleo, el módulo que se cargaría sería el contenido en el binario de Node, o sea, el módulo del core.

2.2.4. Módulos de terceras partes

Node, además de por las características que se están desgranando, es una gran plataforma de desarrollo por el inmenso ecosistema que ha crecido en torno suyo. Se pueden encontrar infinidad de módulos desarrollados por terceras partes para usar en proyectos propios, o desarrollar librerías con un propósito específico, tanto para uso propietario como para uso de la comunidad Node.

[16] IEEE: *Institute of Electrical and Electronics Engineers*

A la hora de utilizar módulos de terceras partes, se debe ser capaz de especificar su ubicación para cargarlos o saber dónde Node va a buscarlos para instalarlos. Cuando se indica, a través de la función del estándar *CommonJS*, que un módulo es necesario, el orden que sigue Node para resolver la ruta del fichero hasta encontrarlo es:

1. Cargar el fichero de nombre igual que el identificador del módulo, `modulo`.
2. Si no lo encuentra, le añadirá la extensión `.js` y buscará `modulo.js` que interpretará como un fichero de código JavaScript.
3. Si tampoco lo encuentra, le añadirá la extensión `.json` y buscará `modulo.json` e interpretará el contenido como un objeto en notación JavaScript (JSON).
4. Por último, buscará `modulo.node` e intentará cargarlo como código C/C++ compilado, a través del método `process.dlopen()` comentado con anterioridad.
5. Si el identificador del módulo fuese un directorio, por ejemplo `modulo/`, Node buscará el archivo `modulo/package.json` que contiene información de cuál es el punto de entrada a la librería. De no hallarlo, intentaría resolver dicho punto se ha descrito antes, primero buscando `modulo/index.js` y, de no encontrarlo, buscando `modulo/index.node`.

Determinar la ruta dentro del sistema de ficheros de la máquina donde se halla el fichero depende de si el identificador de módulo comienza por:

- / se está referenciando una ruta absoluta en el sistema, con lo que el cargador empezará a buscar desde la raíz del sistema de archivos del sistema operativo
- ./ o ../ la referencia es a una ruta relativa al directorio donde se encuentra el módulo
- si el identificador no comienza con las barras, ni es un módulo del core, Node lo buscará en el directorio *node_modules/* que lo supone al mismo nivel en el árbol de directorios que el fichero de código que requiere al módulo. Si no lo encuentra ahí, irá buscándolo en todos los directorios padres hasta llegar al directorio raíz del sistema de ficheros. El propósito de ello es localizar las dependencias de una aplicación para que no colisionen.

Por último, existen unas variables de entorno desde donde Node buscará librerías en caso de que todo lo anterior falle. Por ejemplo, desde la ubicación que indique `NODE_PATH`. No obstante, este método se desaconseja totalmente por haber quedado obsoleto, con lo que no se le va a dar mayor importancia.

2.3. "Construida encima del entorno de ejecución de JavaScript de Chrome"

2.3.1. El lenguaje JavaScript

El lenguaje de programación que se usa en Node es Javascript. Javascript es un dialecto de la especificación estándar ECMA-262 [12]. El uso mayoritario ha estado en el llamado "lado del cliente", en referencia a la arquitectura cliente-servidor, para la capa de presentación de las aplicaciones web, íntimamente ligado con el *DOM (Document Object Model)* [13] que es la interfaz de programación estándar para acceder y manipular el contenido, la estructura y el estilo de los componentes de una página web con los que JavaScript interactúa en el Navegador proporcionando una experiencia interactiva al usuario.

Es un lenguaje que se caracteriza principalmente porque es: [14]

- interpretado: está diseñado para que una capa intermedia de software, el intérprete, lo ejecute, en contraposición a los lenguajes compilados, que corren directamente sobre la arquitectura y sistema operativo objetivo (por regla general, ya que Java es compilado pero se ejecuta dentro de una máquina virtual)

- dinámico: ligado con la característica anterior, realiza acciones que en otro tipo de lenguajes se harían en tiempo de compilación, como evaluación de código, o *eval*, que es la interpretación de código en tiempo de ejecución, o como la modificación, también en tiempo de ejecución, de las características de las clases o el tipo de las variables

- funcional: la programación funcional es un paradigma de programación que se basa en la evaluación de expresiones, como son las funciones, evitando tener estados y datos mutables, a diferencia de la programación imperativa que se basa en cambios de estado a través de la ejecución de instrucciones

[15]. JavaScript posee características de la programación funcional como son las Funciones de Orden Superior. Se denominan así a las funciones que admiten por parámetro otras funciones o que como resultado devuelven una función.

- orientado a objetos parcialmente (o basado en objetos): de los tres requisitos básicos que definen este tipo de programación, JavaScript no soporta el polimorfismo y la encapsulación sólo es posible para funciones dentro de funciones (las funciones de orden superior antes mencionadas), aunque posee un modelo de herencia por prototipado.

- débilmente tipado: se realiza una conversión, o *cast*, del tipo de las variables en tiempo de ejecución, según el uso que se hace de ellas.

JavaScript encaja perfectamente en el paradigma de la programación orientada a eventos, en la que el flujo del programa no es secuencial sino que depende de los eventos, asíncronos en su mayoría por naturaleza, que se producen durante la ejecución del mismo. Las características funcionales del lenguaje de las que hemos hablado son útiles en este paradigma. Las funciones de primer orden hacen posible el uso de:

- funciones anónimas: son las funciones que se pasan como parámetro a las funciones de orden superior. Son útiles para usarlas como *callback*, asignándolas por parámetro a algún objeto mediante el método correspondiente. Los *callbacks* son funciones que se ejecutan como respuesta a un evento.

- *closures*, o cierres: se generan cuando una variable queda referenciada fuera de su *scope* a través, por ejemplo, de una función devuelta por otra función de orden superior. Se emplean en *callbacks* para hacer referencia a *scopes* que de otra manera se perderían. [16]

Al ser un lenguaje interpretado, Javascript requiere de un intérprete o máquina virtual que lo ejecute. En la actualidad hay varios de estos intérpretes disponibles, por ejemplo: SpiderMonkey de Mozilla, Nitro de Apple o V8 de Google.

SpiderMonkey y V8 fueron diseñados para servir como motor Javascript de los navegadores web Mozilla Firefox y Google Chrome respectivamente. Pero, a diferencia del motor Nitro para Webkit, pueden ser embebidos en otras aplicaciones, como es este caso.

2.3.2. El motor V8 de Google

La elección del motor V8 de Google para Node se debió a que ambos proyectos comenzaron prácticamente a la vez [3] y, en palabras del autor de Node, "V8 es una buena, organizada librería. Es compacta e independiente de Chrome. Se distribuye en su propio paquete, es fácil de compilar y tiene un buen archivo *header* con una buena documentación. No tiene dependencia de más cosas. Parece más moderna que la de Mozilla" [17]. Además este motor presenta unas características revolucionarias a la hora de interpretar JavaScript.

Para realizar una comparativa entre los diferentes motores JavaScript se pueden ejecutar los test de *benchmarking* [18] diseñados para probar V8. Estas pruebas evalúan el rendimiento en operaciones matemáticas complejas, como criptografía, o una simulación de kernel de sistema operativo entre otros muchos, y se ejecutan sobre el motor presente en el navegador.

V8 es de código abierto, bajo licencia New BSD [19], y está escrito en C++. Implementa la 5ª edición del estándar ECMA-262 y es multiplataforma, lo que ha permitido a Node estar presente en sistemas tipo Unix (estándar POSIX), Mac y, a partir de la versión 0.6, en Windows [20].

Dos de las características principales de V8, que lo han hecho destacar sobre el resto de motores, son:

- compilación y ejecución de código JavaScript: el código fuente se pasa código máquina cuando es cargado y antes de ser ejecutado por primera vez.
- recolección eficiente de basura: este término se utiliza al hablar de la liberación de memoria que ocupan los objetos JavaScript cuando no van a usarse más. V8 emplea un recolector *stop-the-world*, es decir, V8, cíclicamente, detiene la ejecución del programa (en la llamada "pausa embarazosa") procesando sólo una parte de la memoria heap para minimizar el efecto de la parada. Durante esta parada no se crean nuevos objetos pero también se evita con ella que no haya indisponibilidad momentánea de los existentes [21].

V8 permite a cualquier aplicación C++ que lo use hacer disponibles sus objetos y funciones al código JavaScript que ejecuta. Un claro ejemplo de esto es el ya comentado objeto `process` disponible en el *scope* global de cualquier programa para Node.

Todo el código JavaScript que corre en Node, tanto el de su núcleo como el de programa, se ejecuta en un único contexto que se crea cuando se inicia la plataforma. Para V8, los Contextos son entornos de ejecución separados y sin relación que permiten que se ejecuten varias aplicaciones JavaScript, una en cada Contexto, en una única instancia del intérprete [22]. En el caso de Node, como se ha comentado, sólo hay uno y sólo una instancia de V8 por lo que únicamente se ejecuta un programa a la vez. Si se desea ejecutar más de un script de Node, se deben levantar más instancias.

El hecho de que la plataforma ejecute una sola instancia de V8 induce a pensar que en máquinas con procesadores de múltiples cores, todos menos uno quedarán desaprovechados. Para paliar esta deficiencia existe un módulo, *cluster*, aún en estado experimental, que permite lanzar una red de procesos Node, en el que uno de ellos hace de maestro (*master*) y el resto de trabajadores (*workers*). El rol de maestro es el de vigilar que los procesos trabajadores ejecuten su tarea, que será la misma en todos, incluso compartiendo puertos TCP (a diferencia de si se lanzasen los procesos mediante el módulo *child_process*). Un error en alguno de los workers que lo lleve a detenerse generará un evento `'death'` que puede ser recogido por el maestro para proceder conforme esté programado. Por supuesto, entre los procesos puede haber comunicación a través de mensajes, con la misma API que se emplea en el módulo *child_process*.

2.4. "Fácil desarrollo de rápidas, escalables aplicaciones de red"

Una ventaja de emplear JavaScript como lenguaje para las aplicaciones en Node es que, al ser un lenguaje con una curva de aprendizaje pronunciada, es decir, que sus fundamentos básicos se aprenden fácilmente, es posible empezar a desarrollar aplicaciones rápidamente con sólo tener unas nociones de las características fundamentales del lenguaje y conocer el grueso de las librerías del núcleo de la plataforma.

Un ejemplo muy típico, y que aparece en la página web de Node, es un sencillo servidor HTTP que responde a las peticiones con un "Hello World" en tan sólo cinco líneas de código:

```
var http = require('http');
```

```
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, "127.0.0.1");
console.log('Server running at http://127.0.0.1:1337/');
```

En el código anterior se puede apreciar lo comentado:

- el uso de `require('http')` según *CommonJS* para indicar que se usa el módulo *http*
- el uso de funciones de ese módulo para tratar con servidores web y peticiones HTTP
- una de las características fundamentales de JavaScript relacionada con su orientación a eventos: el *callback*, que es la función que se pasa como parámetro a `createServer()` y que se ejecutará cada vez que el servidor reciba una petición HTTP

Este trozo de código se ha usado para realizar pruebas de rendimiento y comparativas frente a otras tecnologías para usos similares que emplean otra aproximación (por ejemplo, *multithreading*) para tratar con los problemas asociados a la escalabilidad de aplicaciones.

Cuando se habla de escalabilidad caben dos perspectivas sobre el tema:

- escalabilidad horizontal: aquella en la que, idealmente, el rendimiento crece linealmente con la adición de nodos al sistema. En este caso, se tendría un conjunto de máquinas, cada una de ellas ejecutando Node de manera sincronizada con el resto. En la práctica Node no incorpora características que permitan realizar esto de una manera concreta e integrada con su arquitectura pero esto no implica que no sea posible, siempre que lo desarrolle el programador.
- escalabilidad vertical: en la que se busca un aumento, también lineal idealmente, de rendimiento añadiendo recursos a un solo nodo, donde corre el sistema. Los recursos principales son la CPU, memoria, disco y red donde a Node puede exigírsele un poco más [23]. Pero la puesta a punto de estos recursos es sólo una parte para obtener el máximo rendimiento.

Una de los aspectos con mayor impacto en la escalabilidad es el diseño del sis-

tema. Éste es uno de los puntos fuertes de Node. Su arquitectura y la forma en que las aplicaciones deben programarse para correr en ella hacen que se cumplan principios básicos del buen diseño para la escalabilidad [24], un par de ejemplos de esto son:

- Sin esperas: el tiempo que un proceso espera a que un recurso esté disponible es tiempo que otro proceso no emplea para ejecutarse. La naturaleza asíncrona y monoproceso de Node aseguran que se optimizará el tiempo de ejecución de ambas tareas.
- Lucha por los recursos: Node gestiona internamente de manera eficiente los recursos del sistema para que todas las operaciones sobre, por ejemplo, red o ficheros que se demandan en el código estén satisfechas sin que se abuse de ellas. Así se evita un uso descompensado de las mismas por las distintas partes del código de la aplicación.

No obstante, la escalabilidad real de Node está todavía por verse [25].

2.5. "Usa E/S no bloqueante dirigida por eventos"

Uno de los puntos críticos, usual cuello de botella, que afecta en alto grado al rendimiento de cualquier sistema, en especial a aquellos que hacen un uso intensivo de datos, ya sea mediante fichero, bases de datos o conexiones a través de una red, son las operaciones de Entrada/Salida con ficheros y dispositivos. Habitualmente, en las presentaciones de Node que su autor emplea para dar a conocer la plataforma, se justifica el diseño de la arquitectura con unas medidas sobre el impacto que tiene la interacción de un sistema normal con las distintas fuentes de datos en términos de latencia y ciclos de reloj de la CPU. En concreto, se manejan las cifras siguientes [26]:

```
caché L1: 3 ciclos
caché L2: 14 ciclos
     RAM: 250 ciclos
   disco: 41.000.000 ciclos
     red: 240.000.000 ciclos
```

Las latencias correspondientes a las cachés del procesador y la memoria RAM son muy bajas, pero se puede considerar que las de disco y red poseen un retardo

excepcionalmente alto que deja al procesador desocupado. La reacción habitual a estos "ciclos muertos", en el sentido de actividad de un programa típico, es, simplemente, la espera a que la operación termine para continuar con la ejecución de las siguientes instrucciones. Es lo que se conoce como "sistemas bloqueantes": durante los ciclos de Entrada/Salida el programa se queda bloqueado en esa tarea.

A nivel de aplicación, el ejemplo común, utilizado también en las presentaciones sobre Node, es la petición típica a una base de datos:

```
result = query('select * from table');
//utilizar result
```

donde no se podrá utilizar el resultado de la petición hasta que no se haya completado. Consecuentemente, se bloquea el proceso entero.

2.5.1. El modelo de Concurrencia de Node

La solución que plantea el autor de Node es un modelo de concurrencia basado en eventos. En este modelo de concurrencia el flujo del programa viene determinado por la reacción del programa a diversos eventos, en su mayoría asíncronos. Las implicaciones que conlleva el uso de un modelo de estas características son varias:

- la necesidad de un bucle de procesado de eventos. Éste se ejecutará en un único proceso y, lo que es más importante, sólo ejecutará un manejador de eventos, o *callback*, a la vez. De esto se desprende que no se debe de ninguna manera realizar operaciones especialmente complejas en ese manejador, que supongan un gran uso de CPU o sean de larga duración puesto que el bucle dejará de atender los siguientes eventos que se produzcan con la consecuente degradación del programa.

- el uso de un lenguaje que encaje en el modelo basado en eventos. En este caso, JavaScript es la opción ideal pues el intérprete de este lenguaje se basa en un modelo idéntico.

La consecuencia más relevante de esto es que el sistema resultante es "no bloqueante": durante los ciclos de Entrada/Salida, el programa realiza las siguientes tareas hasta que se hayan completado dichos ciclos lo cual será notificado con

un evento que desencadenará la ejecución del *callback*, cuyo código modela la siguiente tarea a realizar una vez se dispone del resultado de las operaciones Entrada/Salida.

Nuevamente, a nivel de aplicación, esta vez el código tendría el aspecto que sigue:

```
query('select * from table', function(result){
  //utilizar result
});
```

donde después de la llamada a `query()` se retorna de inmediato al bucle de eventos para seguir con la ejecución del programa. Una vez se haya completado la petición a la base de datos, se ejecutará la función de *callback* en la que se puede utilizar el resultado de la llamada.

2.5.2. Arquitectura de Node

Implementando este modelo se cumple uno de los objetivos de diseño [26] de Node que es que "ninguna función debe realizar Entrada/Salida directamente". Node delega todas estas operaciones a mecanismos del sistema operativo a través de la librería *libuv* [27], desarrollada específicamente para la arquitectura de Node [28]. *Libuv* surge cuando se porta la plataforma a Windows y el propósito es el de abstraer todo el código que sea dependiente de una plataforma específica (*nix/Windows), en concreto aquel referente a la gestión de eventos y operaciones Entrada/Salida no bloqueantes.

En versiones de Node previas, hasta la 0.6, en los sistemas *nix aquella funcionalidad residía en las librerías *libev* y *libeio* respectivamente, y en sistemas Windows, en IOCP. Con la versión 0.6 se introdujo *libuv* que implementaba para Windows la funcionalidad que antes correspondía a IOCP, aunque para *nix seguía teniendo debajo a *libev* y *libeio*. Con la versión 0.8 de la plataforma, *libuv* asume las funciones de las dos librerías para todos los sistemas operativos.

Para entender qué funciones asume *libuv* se debe conocer qué papel desempeñaban las dos librerías que originariamente formaban parte de Node:

- *Libev*, en palabras de sus creadores, es un bucle de eventos en el que se registran los denominados *event watchers* que indican qué eventos debe el

bucle gestionar y cómo debe reaccionar ante ellos, vía *callback*. Los eventos son de muchos tipos, desde que un descriptor, como un fichero o un socket, puede comenzar a leerse hasta que se ha producido un *timeout*. Node de por sí, al arranque, registra muy pocos *watchers*[17], tan sólo los referentes al recolector de basura y aquel que se encarga de la ejecución de funciones postpuestas para la siguiente iteración del bucle [11].

- Por su parte, *libeio* provee los mecanismos de Entrada/Salida asíncrona para conseguir programas completamente no bloqueantes. Ofrece versiones asíncronas de la mayoría de funciones del estándar POSIX para manejo de ficheros, que es para lo que se utiliza en Node quedando la Entrada/Salida de red gestionada por *libev*. Internamente, emplea un *pool* de *threads* [29] que, por la naturaleza de Node, es transparente para el programador y por tanto, no hacen que Node sea *multithreaded* a nivel de sistema, no de aplicación. Además se integra a la perfección en cualquier bucle de eventos, como por ejemplo con el que proporciona *libuv* [30].

El último paso al inicio de Node es la llamada a la función que arranca el bucle de eventos. En este punto ya tenemos cargado en el Contexto todo el código JavaScript, tanto del núcleo (*bootstrap*) como propio, donde se han definido ante qué eventos reacciona el programa y el comportamiento ante ellos a través de funciones de *callback*. La misión del bucle es esperar a que éstos ocurran y ejecutar el código que modela dicho comportamiento.

Desde que se define un *callback* como respuesta a alguna operación de Entrada/Salida, hasta que dicho *callback* entra en ejecución, en Node se siguen una serie de pasos:

1. se carga el módulo que a su vez carga el *binding* correspondientes
2. se instancia la clase que provee el *binding*, disponible a través de V8, a la que se le instalan los *callbacks*.
3. el *binding* escuchará los eventos que se produzcan en las operaciones de Entrada/Salida
4. si hay un *callback* instalado, el *binding* lo ejecutará a través de `MakeCallback()`, una función interna de `node.cc` para ejecutar código JavaScript desde código C.

[17] `node.cc` https://github.com/joyent/node/blob/v0.8.20-release/src/node.cc#L2820

Los *bindings* son la manera que tiene el código en JavaScript de los módulos del core de hacer uso del código nativo [31]. Un *binding* es un módulo escrito en C/C++ que es capaz de utilizar las librerías, por ejemplo *libev* y *libeio*, a través de *libuv*, para interactuar con el bucle de eventos y la Entrada/Salida directamente, que es algo que el código JavaScript no puede hacer. Se podría decir que es una librería a bajo nivel. El *binding* a su vez proveerá una serie funciones al módulo para que éste ofrezca en su `exports` al desarrollador de aplicaciones su funcionalidad. Por ejemplo, los siguientes *bindings* se emplean en las librerías del core de Node:

buffer

Soporte de codificación en diferentes sistemas y manejo de datos binarios a bajo nivel.

Se usa en el módulo `buffer.js`

cares_wrap

Consultas asíncronas a DNS a través de la librería *cares* de la que Node depende.

Se usa en el módulo `dns.js`

constants

Exposición directa de las constantes posibles de la plataforma.

Se usa en los `módulos child_process.js`, `constants.js`, `fs.js`

crypto

Capacidades criptográficas y de cifrado a través de la librería openSSL.

Se usa en los módulos `crypto.js`, `tls.js`

evals

Acceso a funcionalidad de máquina virtual V8 sobre la manipulación de Contextos para la evaluación de scripts con código JavaScript. De ahí el nombre del módulo, `node_script.cc`.

Se usa en los módulos `module.js`, `vm.js`

fs

Operaciones propias del sistema de ficheros, no operaciones con ficheros como el nombre del código fuente del módulo podría indicar (`node_file.cc`).

Entre otras: manejo de directorios o cambio de permisos.

Se usa en los módulos `fs.js`, `path.js`

fs_event_wrap

Modela la clase evento de sistema de archivos que se emplea para notificar cambios o errores en el mismo.

Se usa en el módulo `fs.js`

http_parser

Usado en conjunto con la dependencia `http_parser` de Node, que extrae del mensaje HTTP toda la información relevante. Sirve de capa intermedia entre ésta y el código JavaScript realizando principalmente la gestión de los *callbacks*.

Se usa en los módulos `http.js`, `querystring.js`

natives

Contiene el código fuente de todas las librerías del core de Node. El código fuente del módulo se genera en tiempo de compilación por lo que no se hallará en disco.

Se usa en el módulo `_debugger.js`

os

Funciones de recogida de información sobre el sistema operativo directamente expuestas a os.js.

Se usa en el módulo `os.js`

pipe_wrap

Capacidades de comunicación con procesos y con *streams* de TCP.

Se usa en los módulos `child_process.js`, `net.js`

process_wrap

Métodos spawn y kill para el manejo de procesos.

Se usa en el módulo `child_process.js`

tcp_wrap

Manejo de conexiones TCP y su *stream* asociado.

Se usa en el módulo `net.js`

timer_wrap

Funciones básicas para el manejo de un *timer*: start, stop, again...

Se usa en el módulo `timers.js`

tty_wrap

Integración con el terminal de *libuv*.

Se usa en el módulo `tty.js`

udp_wrap

Manejo de UDP.

Se usa en el módulo `dgram.js`

zlib

Integración del proceso de compresión y/o descompresión en el *thread pool* de *libuv*.

Se usa en el módulo `zlib.js`

Un módulo cargará un *binding* gracias a la función `binding()` del objeto `process`. Esta función se pensó para uso interno de Node, pero es accesible al programador. Se podrían desarrollar módulos para sustituir a los existentes, si se quisiera o se tuviera la necesidad y conocimientos suficientes.

Cuando un *binding* se carga, su nombre aparece en la lista de módulos cargados `moduleLoadList` del objeto `process` precedido de la palabra *Binding* para diferenciarlo de los módulos JavaScript, a cuyo nombre precede la palabra *NativeModule*.

2.5.3. La clase EventEmitter

Pero la Entrada/Salida no es la única manera que existe en Node de que se produzcan eventos. Se puede dotar a cualquier clase de la habilidad de emitirlos gracias a la clase *EventEmitter* que el módulo *events* facilita y de la que se pueden heredar sus métodos empleando la función `inherits()` del módulo *utils*. Esta parte no hace uso del bucle de eventos, y se produce a nivel de código JavaScript [32].

Básicamente, al proporcionar a una clase métodos para la gestión de eventos, se facilita que:

- la clase dé la oportunidad a otras clases de "escuchar" sus eventos subscribiéndose a ellos a través de métodos como `on()` o `addListener()`, en desuso a favor del anterior. Se puede hacer incluso que un subscriptor escuche un evento una sola vez con `once()`. Es importante indicar que, para evitar posibles pérdidas de memoria, la clase avisa por consola de que se ha superado el límite de 10 subscriptores, aunque permite tenerlos. Sin embargo, como no todas las clases pueden tener esa limitación, se ofrece la posibilidad de aumentarlo siempre que, explícitamente, se fije con una llamada a `setMaxListeners()`

- la clase dé la oportunidad a sus subscriptores de dejar de atender a los eventos que escuchan con `removeListener()`. Incluso se pueden eliminar todos con `removeAllListeners()`

- por supuesto, la clase disponga de un mecanismo para emitir los eventos cuando el programador lo crea conveniente. Este método es la función `emit()` a la que se le indica qué tipo de evento se genera. `emit()` hace uso del objeto `_events`, propiedad de *EventEmitter*, en el que cada propiedad tiene el nombre de un evento y el valor asociado a éstas es un array de *callbacks* instalados por los subscriptores. Por ejemplo:

```
_events = {
  "connect":       [function(){}, function(){}],
  "customEvent":   [function(){}, function(){}],
  //si solo hay un subscriptor no se mete en un array
  "newListener":  function(){}
}
```

2.5.4. Postponiendo la ejecución de funciones

Además de todo lo comentado se dispone de una última forma de interactuar con el bucle de eventos. Se puede programar la ejecución de una función en la siguiente iteración del bucle pasándosela como argumento a la función `nextTick()` del objeto `process`, que es el mecanismo que Node proporciona con tal propósito.

`process.nextTick()` se define en el proceso de *bootstrap* de `node.js` [18]. A través de ella se podrán encolar una o más funciones en `nextTickQueue`, un array no accesible a través del *scope* porque está protegido[19] mediante un cierre, o *closure*, al ser una variable local. El contenido de esta cola se ejecuta en la siguiente iteración del bucle de eventos proporcionado por *libuv*. El funcionamiento interno de este proceso se describe a continuación:

1. `nextTick()` encola el *callback* en `nextTickQueue` y notifica a Node que hay tareas pendientes para la siguiente iteración llamando a `_needTickCallback()` del objeto `process`. Esta función es la única que no se asigna durante el *bootstrap* sino en el arranque de la plataforma[20].

2. el bucle de eventos, al que, en el arranque de Node en `src/node.cc`, se ha programado para que cada una de sus iteraciones ejecute la función nativa `Tick()`[21], comprueba el flag de *callbacks* pendientes `need_tick_cb` y si los hay, invoca a la función JavaScript `_tickCallback()` de la variable global `process` definida en `src/node.js`.

3. `_tickCallback()` va ejecutando las funciones encoladas en `nextTickQueue` conforme las va extrayendo de la cola.

2.6. "Es ligero y eficiente" [1]

Node es una fina capa de software entre el sistema operativo y la aplicación escrita para la plataforma porque los objetivos que se han perseguido con su arquitectura son la velocidad y la eficiencia.

Centrados en esos propósitos, se desecha emplear un modelo *multithread* para manejar las conexiones pues el coste en términos de tiempo para crearlos y memoria consumida por cada uno de ellos es elevado. Se busca entonces una solución de alto rendimiento que se caracterizará por cumplir las condiciones generales para este tipo de aplicaciones: realizar operaciones de Entrada/Salida no bloqueantes delegando en el sistema operativo (a través de *kqueue*, *epoll*...) y coordinarlo todo a través de uno o varios bucles de eventos. La finalidad es multi-

[18] node.js https://github.com/joyent/node/blob/v0.8.20-release/src/node.js#L254
[19] node.js https://github.com/joyent/node/blob/v0.8.20-release/src/node.js#L230
[20] node.cc https://github.com/joyent/node/blob/v0.8.20-release/src/node.cc#L2303
[21] node.cc https://github.com/joyent/node/blob/v0.8.20-release/src/node.cc#L260

plexar la mayor cantidad posible de operaciones de Entrada/Salida en un mismo *thread* en lugar de una por *thread*. Como ejemplos de alternativas que cumplen estas condiciones y pueden ser tomadas como referencia están los *green threads* y los "procesos" del lenguaje Erlang.

Finalmente el modelo escrito en C en que se basa la solución adoptada por Node, como ya se ha desgranado, es un thread que ejecuta un bucle de eventos con Entrada/Salida no bloqueante capaz de manejar alrededor de 20000 flujos Entrada/Salida.

2.7. "Perfecto para aplicaciones en tiempo real *data-intensive*"

2.7.1. Tiempo real y Node

El modelo de concurrencia de Node encaja con los requisitos que se exigen a las aplicaciones en tiempo real flexible (*soft real-time*). Un sistema de tiempo real es un sistema informático que interacciona repetidamente con su entorno físico y que responde a los estímulos que recibe del mismo dentro de un plazo de tiempo determinado [33]. Hay principalmente dos tipos de estos sistemas:

- sistemas de tiempo real flexible, entendiéndose por éstos aquellos en los que las restricciones/condiciones de latencia son flexibles: se pueden perder plazos, es decir, la respuesta al estímulo no cumple las condiciones de tiempo impuestas, y además el valor de la respuesta decrece con el tiempo pero no acarrea un desenlace fatal
- sistemas de tiempo real estricto, en contraposición a los anteriores, donde todas las acciones deben terminar en el plazo especificado y cualquier incumplimiento de las condiciones de retardo puede desembocar en un fallo total del sistema [34].

Un pequeño apunte sobre el tiempo real y el recolector de basura de V8, tal y como se recuerda en [35]: el motor de Google puede modificar la respuesta de un programa, por ejemplo un servidor, a los eventos que se generan, ya que, como se ha señalado anteriormente, este recolector es *stop-the-world*, con lo que se detendrá la ejecución del programa hasta que se realice el ciclo de recolección.

No obstante, un ciclo de recolección no comienza aleatoriamente: es Node quien avisa a V8 de que puede entrar en acción.

Cuando se inicia la plataforma, en la preparación del bucle de eventos, se programa un chequeo para cada iteración del bucle, mediante `node::Check()`, que hará que se ejecute el recolector, mediante `node::Idle()`, si el bucle no tiene ningún evento mejor que procesar. La condición para ello es que la diferencia de tiempos en las últimas iteraciones del bucle sea mayor de 0.7 segundos, si bien es cierto que hay un temporizador, que se ejecuta cada 5 segundos, que realiza también chequeos de memoria y tiempos de inactividad con el mismo propósito. No obstante, estos detalles de implementación están sujetos a cambios, como se puede apreciar en la versión de `src/node.cc` de la rama inestable 0.9.

2.7.2. ¿Para qué es útil Node entonces?

Node es útil por tanto para aplicaciones no críticas que admitan un cierto retardo. Además se debe tener en cuenta su capacidad de manejar un alto número de conexiones y procesar un enorme número de operaciones de Entrada/Salida muy rápidamente. Éste es un requisito especialmente importante para las aplicaciones que hacen un uso intensivo de datos (*data-intensive applications*) porque emplean la mayor parte del tiempo en realizar este tipo de operaciones. Se puede afirmar por tanto que Node encaja de manera excelente si se quiere [35]:

- Interfaces ligeros *REST*/*JSON*: el modelo de Entrada/Salida no bloqueante, para atender las peticiones *REST*, en combinación con JavaScript, para el soporte nativo *JSON*, lo hacen óptimo como capa superior de fuentes de datos como bases de datos u otros servicios web.

- Aplicaciones monopágina: las aplicaciones monopágina son aquellas que se presentan en una única página web, emulando a las aplicaciones de escritorio. La interacción del usuario con el servidor a través de la interfaz de la aplicación se realiza mediante peticiones *AJAX*, en lugar de recargar la página entera. La interfaz se actualizará de acuerdo a dicha interacción, sin abandonar la página. El uso de *AJAX* puede propiciar una avalancha de peticiones que el servidor debe ser capaz de procesar. Es aquí donde Node entra en acción.

  Se debe destacar la ventaja de compartir código, por ejemplo de validación,

entre el cliente y el servidor.

- Reutilizar herramientas de Unix: La capacidad de Node de lanzar miles de procesos hijos que ejecuten comandos y tratar su salida como *streams* permiten utilizar la plataforma para reutilizar el software existente en lugar de reinventarlo para ella.
- Datos por streaming: al tratar las conexiones HTTP como *streams*, se pueden procesar ficheros al vuelo, conforme se envían o reciben.
- Comunicación: por las características comentadas, aplicaciones de mensajería instantánea o web en tiempo real, e incluso, juegos multijugador.

Capítulo 3

Módulos Buffer y Dgram

3.1. Aspectos de UDP relevantes para Node

Las siglas UDP (*User Datagram Protocol*) se refieren a un sencillo protocolo que pertenece al conjunto de protocolos de Internet. Se sitúa en el nivel de transporte, por encima del nivel de red empleándose por ello en comunicaciones punto a punto en redes de conmutación de paquetes basadas en IP. En estas redes el paquete se denomina *datagrama* y se encamina sin que exista un circuito virtual establecido sino que se hace en base a las direcciones que dicho *datagrama* lleva en su cabecera.

Tiene una importante característica: es no fiable. Esto significa que no garantiza ni el orden (no hay control de flujo), ni la entrega (confirmación o ack) y ni siquiera que los *datagramas* no lleguen duplicados. Es por ello que se dice que el protocolo es no orientado a conexión sino a transacción. Si se desea que se cumplan uno o varios de estos puntos, se debe proveer los mecanismos en una capa superior, lo que en la práctica significa que debe implementarlos el programador: UDP simplemente proporciona una manera inmediata de intercambiar *datagramas*.

Es un protocolo extremadamente sencillo lo que en la práctica significa que es muy ligero, es decir, apenas añade bytes adicionales a los que se quieren enviar y evita la necesidad de una fase de inicialización de la conexión. Consecuentemente se dispondrá de muy pocos parámetros que se puedan ajustar a la hora de realizar comunicaciones con él.

La cabecera tiene cuatro campos de 16 bits cada uno (2 octetos):

- puerto de origen (opcional)
- puerto de destino
- longitud del *datagrama* completo (cabecera y datos): indica el número de octetos total de la cabecera más los datos. Su mínimo valor es 8 por tanto (cuando sólo se computa la cabecera) y deja para los datos un espacio de aproximadamente 64K octetos.

  En la práctica los *datagramas* no son tan grandes. Existe una limitación impuesta por el parámetro MTU del nivel de red. MTU (*Maximum Transfer Unit*) indica el máximo número de octetos que puede tener un *datagrama* contando las cabeceras. Se tiende por tanto a que los *datagramas* que se generan se ajusten a dicha MTU. Cabe la posibilidad de que el nivel IP fragmente los que excedan la MTU, pero en la práctica no se hace puesto que ensamblarlos requiere de un mecanismo que verifique pérdidas de *datagramas* completos y orden de segmentos, y dicho mecanismo correspondería a un nivel superior [36].
- *checksum* (opcional), para verificar la integridad de los datos.

El resto del *datagrama* lo compone el mensaje, en forma binaria, que, junto con el puerto de destino, es la información indispensable que necesita este nivel.

El conjunto de aplicaciones basadas en UDP es, en principio, bastante limitado. La RFC de 1980 hace referencia a dos: al Servicio de Nombres de Internet (DNS, *Internet Name Server*) y al Transferencia de Archivos Trivial (TFPT, *Trivial File Transfer*). Posteriormente y con la aparición de tecnologías de audio y vídeo en tiempo real, donde los requisitos de rapidez y bajo retardo se imponen a los de fiabilidad y orden en la entrega, cosa que ajusta muy bien a lo que ofrece UDP, lo han hecho candidato ideal como base del Protocolo de Tiempo Real (RTP, *Real Time Protocol*).

Íntimamente relacionado con RTP, y la distribución de contenido multimedia en general, aparte de, por supuesto, otro tipo de aplicaciones, están los conceptos de difusión, tanto *broadcast* como *multicast*. Ambos son formas de distribución de datos punto-multipunto o multipunto-multipunto, que es justamente lo que puede exigirse a un servicio de distribución de audio/vídeo. Esta difusión, sin embargo, no se realiza a nivel de transporte, sino que se hace a nivel de red, mediante el protocolo IP y las posibilidades de direccionamiento que ofrece.

Una de las maneras de enviar tráfico a los clientes es el denominado *Broadcasting*. Con este tipo de direccionamiento, el servidor envía datos a todos los clientes de la red y ellos son los que deciden si estos datos son de su interés o no. Como es lógico no se envían cliente por cliente, ya que supone un desperdicio de ancho de banda, sino a la dirección de *broadcast* de la red.

Como se sabe, en una red hay dos direcciones especiales: aquella en la que los bits del sufijo son todo ceros, que identifica a la red, y aquella en la que los bits del sufijo son todo unos. Ésta última es la dirección de *broadcast* y todos los interfaces, aparte de escuchar la dirección IP que tienen configurada, escuchan el tráfico dirigido a esta otra IP.

El otro modo de envío de tráfico a múltiples puntos es el *Multicasting*. Se emplea cuando el número de nodos interesados en unos datos determinados es lo suficientemente grande como para no enviarlos uno a uno, pero lo suficientemente pequeño como para que no haya necesidad de hacer *broadcast*. Como en el caso anterior, también se emplean direcciones IP especiales.

En multicasting se trabaja con direcciones IP de un rango muy concreto, el que va de `224.0.0.0` a `239.255.255.255`. Sin embargo, IANA (el organismo que controla la asignación de estas direcciones), es muy restrictivo con ellas y otorga generalmente direcciones de los rangos `224.0.0.0 - 224.0.0.255`, `224.0.1.0 - 224.0.1.255` y `224.0.2.0 - 224.0.2.255` [37]. Cada una de las direcciones de estos rangos representa un conjunto de máquinas a las que los *routers* dirigen el tráfico. Se dice que estas máquinas están suscritas al grupo *multicast*, por tanto debe existir la manera de indicarle a la red la suscripción y el abandono de un grupo.

3.2. UDP en Node

Una aplicación para Node puede hacer uso de UDP importando la librería *dgram*:

```
var dgram = require('dgram');
```

La manera de obtener un socket UDP es sencilla, tanto para cliente como para servidor, es invocando el método:

`dgram.createSocket(tipo, [function(mensaje, rinfo)])`

al que se pasa como parámetro el `tipo` de soporte IP: `udp4` para IPv4 y `udp6`

para IPv6. Opcionalmente y para sockets servidor, la función de *callback* será la que atienda los mensajes que lleguen de los clientes instalándose como *listener* del evento `'message'`, por lo que puede hacerse más adelante.

Como librería que implementa un protocolo asociado al nivel de transporte, ofrece métodos que cubren las primitivas de dicho nivel:

`dgram.send(mensaje, offset, longitud, puerto, direccion, [function(error, bytes){ }])`

Envía un *datagrama* que contiene un `mensaje` de tipo *Buffer* de `longitud` determinada a un `puerto` de una `direccion` IP. Si sólo se envía una parte del *Buffer*, `offset` indicará en qué posición empiezan los datos a enviar dentro de él, siendo 0 en el caso de enviarse entero.

`dgram.bind(puerto, [direccion])`

Permite escuchar en un `puerto` la llegada de *datagramas*. En máquinas con varios interfaces, se escuchará en todos excepto si se especifica uno concreto pasando como argumento su `direccion` IP.

Si la llamada a esta función tiene éxito, se emite el evento `'listening'`.

```
socket.on('listening', function(){  });
```

`dgram.close()`

Deja de escuchar en el socket y lo cierra, emitiendo el evento `'close'`. A partir de que se emita este evento no se generarán más eventos `'message'`.

```
socket.on('close', function(){ ... });
```

Para la recepción de datos se emplea un *callback* que atiende al evento `'message'`, emitido cuando se recibe un *datagrama* cuyo *payload*, `msg` de tipo *Buffer*, e información del origen, `rinfo`, se pasan como argumentos a dicho *callback*:

```
socket.on('message', function(msg, rinfo){  });
```

Adicionalmente, `rinfo` indica el número de bytes del *datagrama* en la propiedad `rinfo.size`. Por ejemplo:

```
rinfo = {
  address: '192.168.21.7',
  family: 'IPv4',
  port: 50251,
```

```
  size: 436 }
```

Junto con los tres eventos vistos (`'listening'`, `'message'`, `'close'`) se encuentra `'error'`, que se emite sólo cuando ocurre un error en el socket:

```
socket.on('error', function(){  });
```

Aparte de los métodos anteriores está disponible una función auxiliar con la que obtener información (dirección, puerto y protocolo) del socket que la máquina está empleando en la comunicación:

`dgram.address()`
: devuelve el objeto en notación JSON, por ejemplo:

```
{ address: '0.0.0.0',
  family: 'IPv4',
  port: 23456 }
```

Más allá del nivel de transporte, Node también permite controlar ciertos parámetros del nivel de red, justamente aquellos relacionados con la difusión de la que antes se ha comentado.

El único parámetro más general que se puede ajustar es el TTL (*Time To Live*). Este tiempo de vida es un entero entre 1 y 255 (8 bits) que marca el número máximo de saltos que un paquete IP puede dar entre redes antes de ser descartado. Típicamente es 64 y se fija con:

```
dgram.setTTL(saltos);
```

Como se ha introducido al principio del tema, UDP está íntimamente relacionado con la difusión a grupos *broadcast* y *multicast*. Mientras que habilitar o deshabilitar el envío a los primeros es tan sencillo como invocar:

```
dgram.setBroadcast(flag);
```

para los grupos *multicast* hay un completo grupo de funciones:

`dgram.addMembership(direccionMulticast, [interfaz])`
: es el mecanismo que el módulo ofrece para la subscripción de la `interfaz` de la máquina a un grupo de *multicast* identificado con la dirección IP `direccionMulticast`. Si no se especifica la `interfaz`, se añadirán todos los posibles.

dgram.dropMembership(direccionMulticast, [interfaz])

realiza la acción opuesta al método anterior, sacando a la `interfaz` del grupo *multicast* de dirección IP `direccionMulticast`. Al igual que con `dgram.addMembership()`, si no se especifica `interfaz` alguna, se sacan del grupo todos los que pertenezcan a la máquina.

dgram.setMulticastLoopback(flag)

habilita o deshabilita a la interfaz local para que también reciba o deje de recibir los paquetes *multicast* que se envían desde el mismo.

dgram.setMulticastTTL(saltos)

establece, de manera idéntica a `dgram.setTTL()`, el número de `saltos` que un datagrama *multicast* puede dar en una red.

3.3. Codificación de caracteres

El propósito final de cualquier protocolo de comunicación, por tanto también de UDP, es la de transmitir datos entre dos o más puntos. Estos datos son datos binarios que pueden representar texto, audio, vídeo, etc...

Cuando se trabaja con texto, o sea, cadenas de caracteres, debe existir una forma de hacer "legible" el conjunto de bits sin formato que existe en memoria agrupándolos de manera que se transformen en texto reconocible, con o sin sentido, por el ser humano. Este proceso se conoce como *Codificación de caracteres*. En él se establece una correspondencia unívoca, llamada esquema de codificación, entre secuencias de bits y caracteres correspondientes a algún alfabeto concreto definido por su "juego de caracteres"(*character set*).

Existe un amplísimo número de esquemas de codificación, muchos con propósito de mejorar y ampliar los anteriores, de tal manera que una misma secuencia de bits determinada puede representarse de distintas maneras según el juego de caracteres que se elija.

Los más habituales son:

ASCII

[38] este esquema está diseñado para la representación de texto usando el alfabeto inglés. Empleando 7 bits para ello, por lo que define 128 caracteres, incluyendo alfanuméricos, símbolos y 33 caracteres de control, como por

ejemplo *NULL* o *CR* (retorno de carro). Hacer notar que un byte consta de 8 bits, de tal manera que si cada carácter se codifica con uno, todavía quedan 128 caracteres por definir. Esto se ha empleado habitualmente para definir *ampliaciones* del código, existiendo multitud de ellas, incompatibles entre sí puesto que sólo tienen en común los 128 primeros caracteres.

Unicode

[39] es la solución propuesta a los problemas aparecidos por el gran número de codificaciones. Pretende emplear un sólo código para todos los alfabetos posibles, siendo capaz de representar más de un millón de caracteres. Introduce los llamados "puntos de código" (*codepoints*) que son, por decirlo de alguna manera, un identificador de carácter. Así, a una "A" le corresponde un *codepoint*, a una "B" otro *codepoint*...Los *codepoints* se distinguen por ser un número hexadecimal que comienza por *U+*. Para el caso anterior, a la "A" le corresponde el *codepoint U+0041*, a la "B" el *U+0042*...JavaScript es un lenguaje de programación pensado para un Entrada/Salida formateada como texto. Y éste codificado en Unicode.

Son varias las maneras en que Unicode realiza las codificaciones de los *codepoints*. La primera de ellas es la más conocida y extendida en uso: UTF-8. Es una codificación multibyte: los primeros 128 *codepoints* se almacenan con un solo byte, y los siguientes se van almacenando con dos, tres o hasta seis bytes.

Otra de ellas, la tradicional, es la UTF-16 o UCS-2, tanto *Big* como *Little Endian*. Emplea 2 bytes (o 16 bits) para codificar cada *codepoint*. El resto de codificaciones no son tan relevantes para la comprensión del API de Node, por lo que se dejan en el aire en este capítulo.

Al margen de la codificación pensada para el alfabeto humano, existen esquemas de codificación que no establecen una correspondencia entre bits y un alfabeto humano determinado sino entre bits y un subconjunto de ese alfabeto. Por ejemplo, para transformar conjuntos de octetos en conjuntos de caracteres de texto imprimibles. Existe un caso concreto, *Base64*, en que los bytes se transforman en el rango de 64 caracteres compuesto por `A-Z`, `a-z` y `0-9` más los símbolos `+` y `/`. Esta codificación se suele usar para adaptar texto en otros lenguajes o contenido binario (audio, video, ejecutables...) a protocolos que sólo admiten una codificación muy concreta y más restringida. Por ejemplo, para enviar un correo electrónico con el protocolo SMTP (descrito en la RFC821) hay definido un formato

de mensaje cuya codificación debe ser ASCII (según la RFC822) [40]. Si se adjunta contenido multimedia o texto en otro lenguaje con símbolos que no pueden representarse en esa codificación, como puede ser un vídeo, habría problemas para su envío o recepción. En este caso concreto, se introdujo un mecanismo, MIME [41], que utiliza Base64 para codificar el contenido de los mensajes de correo y hacer posible su correcto intercambio.

El proceso de codificación en *Base64* consiste, a grandes rasgos, en coger los bytes de tres en tres, 24 bits en total. Agrupándolos de seis en seis bits, se obtienen cuatro índices, comprendidos entre 0 y 63 (2^6 = 64), con los que se indexa la cadena compuesta por todos los rangos de caracteres `ABCDEFGHIJKLMOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/` de donde se obtiene el carácter buscado. Hay que observar que cada uno de estos caracteres es de 8 bits obtenido a partir de cuatro grupos de 6 bits. Por tanto, cada tres bytes, se generan cuatro.

Otro de los casos es la representación el Hexadecimal: representa los dígitos binarios en notación hexadecimal. Puesto que el sistema de numeración en base 16 emplea los números 0-9 y las letras A-F cada octeto estará representado por combinaciones de dos de estos caracteres.

3.4. Buffers en Javascript

Hay ocasiones en las que la entrada o salida del sistema no es texto sino datos binarios como, por ejemplo, los que se reciben en las conexiones UDP o TCP si se está trabajando con redes, o los que se reciben a raíz de la lectura de un fichero, si se trabaja con el sistema de ficheros de la máquina donde se ejecuta el programa. En estos casos se necesita tratar con los octetos *en bruto*, tal y como están en memoria, accediendo a ellos y transformándolos o convirtiéndolos en determinados tipos de datos según sea necesario durante la ejecución del programa.

Con este propósito existe el objeto *Buffer*: tener una manera eficiente de crear, manipular y consumir bloques de octetos. A bajo nivel, el API de Node habla de que un objeto *Buffer* contiene una zona de memoria tal cual está alojada fuera del *heap* del motor V8. Puede verse como un array de bytes con tamaño fijo que no puede redimensionarse.

El módulo *buffer*, que es el que provee el objeto *Buffer*, está presente en el espacio global de ejecución del script `src/node.js`, con lo cual no es necesario importarlo explícitamente con `require()`. Directamente, se puede instanciar un *Buffer* de tres maneras:

`var buf = new Buffer(size)`
: Con `size` se especifica un tamaño en octetos fijo, que luego no podrá modificarse.

`var buf = new Buffer(array)`
: Se crea un *Buffer* a partir de un `array` de octetos.

`var buf = new Buffer(str, [codificacion])`
: Crea un *Buffer* que contiene el String `str` indicando opcionalmente su `codificacion`. Las codificaciones válidas son: `ascii`, `utf8` (la que se emplea por defecto), `utf16`, `base64` o `hexadecimal`.

A partir de este punto tenemos un *Buffer*, `buf`, del que conocemos su tamaño con `buf.length`, sobre el que se pueden realizar operaciones de escritura y lectura.

Para escribir en un *Buffer* se puede optar por:

- una indexación directa, como con los arrays, `buf[indice]`. De esta manera se accede a la posición `indice` cuyo octeto se puede fijar con un valor que, al tener 8 bits, varía entre 0 y 255.

- el método `buf.write(string, [inicio], [longitud], [codificacion])` que permite escribir `longitud` bytes a partir de la posición `inicio` del *Buffer* la cadena de caracteres `string` codificada según `codificacion` (por defecto `utf8`). Devuelve el número de octetos escritos.

  Se debe tener en consideración que la longitud de la cadena puede no coincidir con el número de bytes escritos puesto que éstos dependen de la codificación empleada. Por ejemplo, hay caracteres en `utf8` que ocupan varios bytes como ya se sabe y se comprueba fácilmente:

  ```
  var bufferedData = new Buffer("Lingüística");

  console.log("String:", data.length, "bytes");
  console.log("Buffer:", bufferedData.length, "bytes");
  ```

También se debe contemplar también el caso que no haya suficiente espacio en el *Buffer*, con lo que sólo se escribirán los bytes que quepan, sin escribir parcialmente un carácter. Esto es, si queda una posición libre de un byte y se pretende introducir un carácter Unicode, que ocupa dos, no será posible.

- introducir el contenido numéricamente, no como caracteres. Así se pueden escribir:
  - enteros de 8 bits con y sin signo, cuyos rangos válidos son, -128 a 127 y 0 a 255, respectivamente, que ocuparán una posición del Buffer
  - enteros de 16 bits con y sin signo, cuyos rangos válidos son, -32768 a 32767 y 0 a 65535, respectivamente, que ocuparán dos posiciones del Buffer
  - enteros de 32 bits con y sin signo, cuyos rangos válidos son, -21474883648 a 21474883647 y 0 a 4294967295, respectivamente, que ocuparán cuatro posiciones del Buffer
  - float de 32 bits
  - double de 64 bits

  Además con los números de más de un byte se puede elegir el formato en el que se almacenan en el *Buffer*: bien *Big Endian*, *BE*, o bien *Little Endian*, *LE*. La diferencia entre ambos está en el orden en que se guardan en memoria.

  Big Endian asigna los bytes más significativos a los índices más bajos del *Buffer* y, por el contrario, *Little Endian* asigna el byte menos significativo a los índices más bajos. Sirva de ejemplo, el entero de 32 bits `0xEF4A71B8` se escribiría en el *Buffer*, representado aquí por `[]`, como `[EF, 4A, 71, B8]` en *Big Endian* y como `[B8, 71, 4A, EF]` en *Little Endian*.

  El motivo de estos dos formatos es histórico: cada arquitectura de microprocesador emplea una u otra (algunas las dos). Por ejemplo, *Sparc* utiliza *Big Endian* y los procesadores de *Intel*, *Little Endian*.

  Teniendo en cuenta todo esto, la clase *Buffer* proporciona 14 métodos para la escritura en formato numérico cuya signature se determina como:

```
write + [U] + [Int | Float | Double] + [8 | 16 | 32] + [BE | LE]
(value, [offset], [noAssert])
```

Las combinaciones posibles deben ser lógicas y de acuerdo a lo explicado antes, por ejemplo, son válidas `buf.writeUInt16LE()` o `buf.writeFloatBE()` y no son válidas, por ejemplo, `buf.writeFloat32LE()`, porque *Float* siempre es de 32 bits y por tanto no se especificaría, o `buf.writeUDoubleBE()`, porque *Double* no es un tipo que se pueda escoger con signo o sin él. Los correctos serían, `buf.writeFloatLE()` y `buf.writeDoubleBE()`.

Por otra parte, para operaciones de lectura, se puede acceder al contenido de un *Buffer* de varias maneras, dependiendo del tipo de dato que se quiera leer y de cómo se quiera hacer esta lectura.

- de nuevo, a través de indexación directa, `buf[indice]` que devuelve el octeto de la posición indice cuyo valor está comprendido entre 0 y 255.
- obteniendo la cadena de caracteres que está representada en el *Buffer* con `buf.toString([codificacion], [inicio], [fin])`. Se puede especificar, de manera opcional, qué `codificacion` debe tener la cadena, y las posiciones de `inicio` y `fin` dentro del *Buffer*. Por defecto, se obtendrá una *String* en `utf8` de todo el *Buffer* (inicio 0 y fin `buf.length`).
- leyendo el contenido numéricamente mediante los métodos duales a los que se emplean para operaciones de escritura. Si en estas operaciones el nombre del método empezaba por *write*, ahora comienza por *read*.

La clase *Buffer* además ofrece operaciones similares a las que se pueden realizar con un array:

`buf.copy(bufferDestino, [inicioDestino], [inicioOrigen], [finalOrigen])`

como su nombre indica, copia en el `bufferDestino` a partir de la posición `inicioDestino`, el contenido del *Buffer* origen, `buf`, desde la posición `inicioOrigen` hasta la posición `finalOrigen`. Por defecto se copia todo el *Buffer* `buf` en `bufferDestino` y en caso de que se intente copiar más cantidad de la que cabe a partir de donde se empiece en destino, sólo se copiarán las posiciones que quepan. Hay que tener en cuenta que un carácter *Unicode* puede ocupar más de una posición, con lo que quedará ilegible si se copia parcialmente.

`buf.slice([inicio], [final])`

es una función que devuelve un *Buffer* cuyo contenido es el mismo que el

contenido del *Buffer* `buf` entre las posiciones `inicio` y `final` pero, muy importante, porque hace referencia a él. Esto significa que si se modifica cualquier octeto en alguna posición de ese rango en alguno de los dos *Buffers*, el cambio se refleja en ambos.

`buf.fill(valor, [inicio], [final])`
asigna el valor deseado a todas las posiciones del *Buffer* `buf` comprendidas entre `inicio` y `final` con un `valor`. Por defecto, se escribe todo el *Buffer*.

`Buffer.concat(lista, [longitudTotal])`
concatena todos los *Buffers* contenidos en el array `lista` devolviendo un nuevo *Buffer*, siempre que `lista` tenga más de uno o no esté vacía, porque si no, devuelve el *Buffer* o no devuelve nada respectivamente.

Es una función muy lenta, que supone un cuello de botella importante en una aplicación, por lo que se debe utilizar sólo si es estrictamente necesario. Una de las maneras para intentar aumentar el rendimiento es pasando el argumento `longitudTotal`, que es la longitud que debe tener el Buffer resultante.

3.5. Aplicación con Buffers y UDP

Como ejemplo del tipo de aplicaciones que se pueden crear con UDP se va a implementar una versión muy sencilla del protocolo RTP. Se empleará para desarrollar una aplicación que emitirá en tiempo real los archivos de audio MP3 alojados en una máquina remota.

3.5.1. Descripción del problema

La aplicación que se pretende realizar es un servidor en tiempo real muy básico de archivos de audio MP3, utilizando RTP. Estos archivos están contenidos en un directorio del servidor y se van reproduciendo en orden secuencial. El destinatario es un grupo de *multicast* definido por la dirección IP `224.0.0.14`, escogida arbitrariamente de entre las posibles, a la que puede unirse como cliente cualquier aplicación de audio con soporte para ello. En concreto, para el desarrollo de la práctica se ha empleado VLC[1].

[1] http://www.videolan.org/vlc/

Se puede configurar de la siguiente manera para escuchar las canciones que emite la aplicación:

1. Ejecutar el Reproductor y en la pantalla principal ir a *Media*

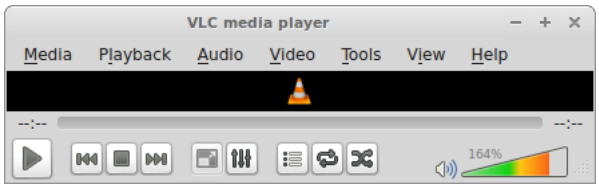

Figura 3.1: Reproductor VLC, Menú “Media”

2. Seleccionar *Open Network Stream* para recibir contenido emitido desde la Red

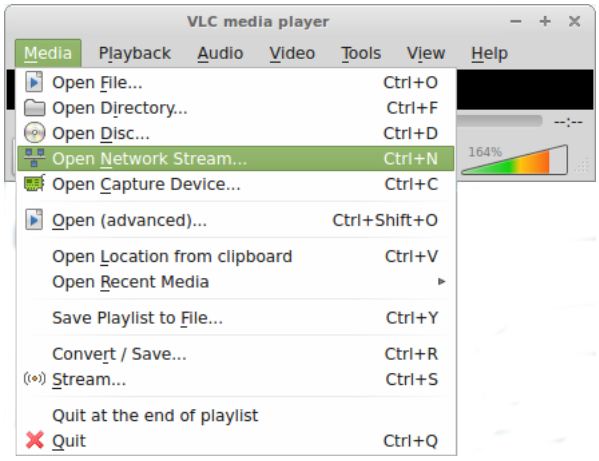

Figura 3.2: Reproductor VLC, “Open Network Stream”

3. Configurar la IP del grupo *multicast* donde se va a recibir la transmisión y en qué puerto. En este caso `rtp://224.0.0.14:5000`, ambos escogidos de manera arbitraria pero coherente con las especificaciones.

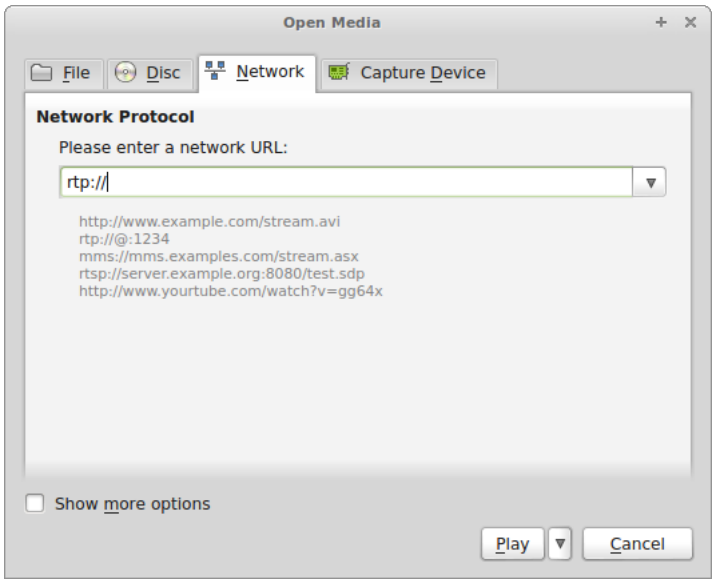

Figura 3.3: Reproductor VLC, configuración de la IP

3.5.2. Diseño propuesto

3.5.2.1. El protocolo RTP

RTP son las siglas de *Real Time Protocol*, un protocolo definido para el envío de contenido multimedia en tiempo real sobre UDP. La versión definitiva de la especificación completa se encuentra en la RFC3350 [42]. Sólo se implementará una pequeña parte de ella, que proporcione la suficiente funcionalidad para el propósito de la aplicación. Esto incluye, básicamente, la generación correcta de la cabecera RTP de los paquetes.

El mínimo contenido funcional de la cabecera de un paquete RTP consta de los siguientes campos, explicados según la RFC propone:

- versión: 2 bits que almacenan el número de versión del protocolo, que va por la segunda, por lo que el valor será siempre 2.
- *padding*: flag que indica la presencia de relleno al final del paquete. Suele ser común en aplicaciones que necesitan un tamaño fijo de paquete. No es el caso de esta aplicación, por lo que su valor puede quedar a false.
- extension: flag que indica que la cabecera se extiende con una cabecera de ampliación que no se va a usar en esta aplicación.

- contador CSRC: la cabecera RTP puede llevar a continuación una lista de fuentes que contribuyen a la generación del contenido del paquete. Este campo de 4 bits indica cuántas van en la lista anexa, pero sólo lo insertan los mezcladores, por lo que carece de utilidad en una aplicación de este tipo.

- *marker bit*: flag cuyo significado depende del contenido que se esté transmitiendo. El propósito general es destacar puntos concretos en la transmisión, como sucede cuando para transmitir un *frame* se emplean varios paquetes. En este caso el *marker bit* activado indica que se ha transmitido el último paquete de los que componen el *frame*.

- tipo de contenido: 7 bits que identifican numéricamente el tipo de contenido que va en el paquete. En este caso, formalmente, es audio *MPEG*, de sus siglas *MPA*, que engloba *MPEG-1* (el tipo de audio empleado en esta aplicación) y *MPEG-2*. La RFC3350 remite a la RFC3551 [43, Section 6] para obtener detalles sobre las características de todos los formatos y el valor del tipo de contenido, en este caso 14.

- número de secuencia: este campo de 16 bits tiene como función la de indicar el orden de los paquetes en la secuencia, útil para poder ordenarlos y detectar pérdidas. Su valor inicial debe ser aleatorio e impredecible.

- *timestamp*: 32 bits cuyo valor indica el instante de muestreo del primer octeto del contenido del paquete. El cálculo de este valor depende del tipo de contenido. En el caso de audio *MPEG*, *MPA*, la RFC3351 marca que la tasa de reloj sea siempre de 90000 Hz, independientemente de la tasa de muestreo del audio (44100 Hz para esta aplicación).

- SSRC: identifica con 32 bits la fuente, único para cada fuente distinta en una sesión. Esta aplicación dispone de una sola fuente, por lo que su valor puede ser establecido sin necesidad de tener en cuenta la probabilidad de colisión con los valores de otras fuentes.

Además de esto, enviar audio en formato MP3 requiere una cabecera adicional específica, tal y como la RFC2250 explica [44, Section 3.5], con dos campos:

- MBZ: 16 bits que no tienen un uso definido y por tanto, no se van a emplear.

- *Fragment Offset*: 16 bits que marcan el inicio del frame de audio dentro de los datos del paquete.

Todo esto se implementa en el módulo *RTPProtocol.js*. Este módulo ofrece la clase *RTPProtocol* cuya misión es simplemente encapsular un *frame* multimedia, en este caso audio MP3 de unas características concretas, en un paquete RTP con la cabecera que corresponda.

3.5.2.2. Descripción de la solución

El resto de piezas de la arquitectura se relacionan como se representan en el siguiente diagrama donde se asocian las clases y los `'eventos'` que emiten con las acciones que se solicitan al resto de las clases:

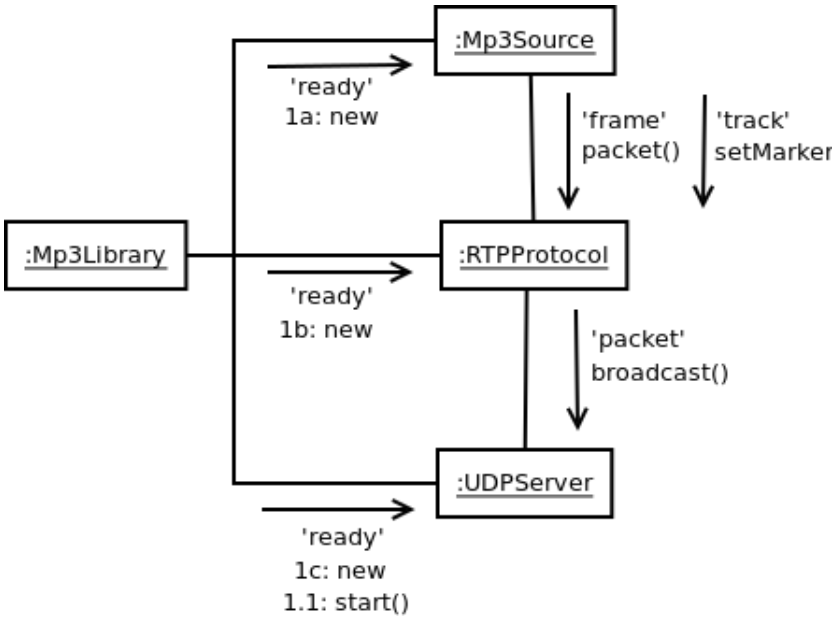

Figura 3.4: Diagrama de Colaboración de la Solución RTP

Se puede comenzar a explicar este diagrama a partir de la clase *MP3Library*, que representa una biblioteca de archivos `.mp3`. Su función es la de buscar los archivos de audio del directorio `./songs/`, dividirlos en sus correspondientes *frames* y almacenarlos en memoria. Pone a disposición del programador métodos para acceder a estos archivos *frame* a *frame*, para que un emisor, como es por ejemplo la clase *MP3Source*, los emita cuando corresponda. Lo único relevante de *MP3Library* para el propósito de la práctica es que emite el evento `'ready'` cuando ha leído y procesado todos los archivos `.mp3` y por tanto está lista para ofrecerlos a las fuentes MP3, *MP3Source*.

La clase *MP3Source* modela el comportamiento de una fuente de audio MP3 que a partir de un archivo genera *frames* con una frecuencia tal que simula que se están emitiendo en tiempo real. En la aplicación que se está desarrollando como ejemplo, sólo se tratará con archivos `.mp3` cuyas características son que están muestreados a una frecuencia de 44100 Hz y codificados con una velocidad de 128 Kbps. Cualquier otro `.mp3` con distintas características producirá un mal-funcionamiento de la aplicación. Estos ficheros `.mp3` se obtienen de la librería *Mp3Library*:

```
var library = new mediaLibraries.Mp3Library;
  library.on("ready", function(){
    var mp3source = new sources.Mp3Source(library);
    // ...
});
```

De *MP3Source* interesa básicamente lo que se ha comentado: que emite el evento `'frame'` a la frecuencia que se emitiría un *frame* si fuera una fuente auténtica de audio en tiempo real. Un cálculo rápido: el estándar MP3 [45] fija por definición que un frame consta de 1152 muestras, y las características que se han elegido para los archivos MP3 determinan que la frecuencia de muestreo sea de 44100 Hertzios. Lo último significa que cada segundo se generan 44100 muestras de tal manera que:

$$\frac{1}{44100}\frac{segundos}{muestras} \times 1152\frac{muestras}{frame} = 26{,}12\frac{mseg}{frame} \tag{3.1}$$

O sea, la clase *MP3Source* emitirá un evento `'frame'` cada 26 milisegundos. Junto con él, se enviará el *frame* para su procesado por parte de los *callbacks* para dicho evento.

El tamaño del *frame* lo determina la segunda condición impuesta a los ficheros de audio, 128000 bits por segundo:

$$\frac{128000}{8}\frac{bytes}{segundo} \times 0{,}02612\frac{segundos}{frame} = 417{,}959\frac{bytes}{frame} \tag{3.2}$$

Es decir, habrá *frames* de 417 bytes, y *frames* de 418 bytes.

No se entrará en más detalles de implementación de estas clases, porque supondría diseccionar casi entero el formato MP3 del que sólo interesa que un archivo de este tipo se divide en *frames*.

Los *frames* que la fuente MP3 emite deben ser encapsulados en un paquete RTP con su cabecera perfectamente formada antes de ser enviados al grupo *multicast*. Por este motivo se hará que el objeto *RTPProtocol* atienda los eventos `'frame'` que emite *MP3Source*.

```
mp3source.on('frame', function(frame){
  rtpserver.pack(frame);
});
```

Cuando a través de estos eventos *RTPProtocol* recibe el *frame* MP3, le añade las cabeceras, con `RTPProtocol.pack(payload)`, y genera un evento `'packet'` para que el objeto en la capa de red que atiende el evento pueda enviarlo. Junto al evento `'packet'` se envía el paquete calculado.

En base a esto, la lógica de *RTPProtocol* es sencilla, sólo requiere crear un paquete RTP de tamaño adecuado `new Buffer(RTP_HEADER_SIZE + RTP_FRAGMENTATION_HEADER_SIZE + payload.length)` e introducir en él los campos como especifica la RFC3350 perfectamente calculados. Notar que la RFC exige que el formato de los campos sea en orden de red (*network byte order*) [42, Section 4], o sea, el byte de mayor peso primero. A esto se le ha denominado en la introducción *Big Endian* y debe tenerse en cuenta a la hora de usar las funciones del módulo *Buffer*:

- la versión, indicadores de *Padding* (que no hay), extensión de la cabecera (que tampoco hay) y el contador CSRC (a cero porque no hay lista con fuentes contribuyentes CSRC) constituyen el primer byte. Son valores fijos en esta aplicación y conforman el número binario 11000000, 128 en decimal:

  ```
  RTPPacket.writeUInt8(128, 0);
  ```

- el siguiente byte lo forman el *Marker Bit* y el tipo de contenido que lleva el paquete, que siempre será el mismo. Para audio MP3, tipo *MPA*, su valor es 14 según la RFC3351. Por tanto, el valor a escribir sólo dependerá del *Marker Bit*: 000000000 si no está activado y 100000 si lo está.

  ```
  RTPPacket.writeUInt8(this.setMarker? 142 : 14, 1);
  this.setMarker = false;
  ```

  Para esta aplicación, que el *Marker Bit* esté activo o no depende de un par de criterios un tanto arbitrarios. El primero es si no hay *payload* por el motivo que sea, por ejemplo que la fuente no haya sido capaz de generar uno en el

tiempo previsto. Se activará siempre después de que se hayan actualizado los campos relativos al orden de los paquetes y al tiempo porque, a pesar de ello, éstos siguen contando.

```
if (!payload) {
  this.setMarker = true;
  return;
}
```

Otro motivo de que se active el *flag* es que se haya empezado a emitir la siguiente canción y se señalice de esta manera al receptor:

```
mp3source.on('track', function(){
  rtpserver.setMarker = true;
}); (Player.js)
```

Una vez reflejado en la cabecera no hay que olvidarse de limpiarlo.

- el número de secuencia, inicializado en el constructor con un valor aleatorio

```
this.seqNum = Math.floor(Math.random() * 1000);
```

se incrementa en uno por cada paquete que se emite:

```
++this.seqNum;
RTPPacket.writeUInt16BE(this.seqNum, 2);
```

- el timestamp, cuyo valor inicial es aleatorio y se calcula en el constructor:

```
this.timestamp = Math.floor(Math.random() * 1000);
```

su valor en cada paquete se incrementa una cantidad igual al tiempo que necesitan para generarse las muestras contenidas en el *frame*, medido con referencia a un reloj de frecuencia 90 kHz. En el caso de audio MPA, según la RFC3551[43, Section 4.5.13], la tasa de incremento del *timestamp* es siempre 90000, independientemente de la tasa de muestreo del audio.

```
var SAMPLES_PER_FRAME = 1152;
var REFERENCE_CLOCK_FREQUENCY = 90000;
var SAMPLING_FREQUENCY = 90000;
var TIMESTAMP_DELTA = Math.floor(SAMPLES_PER_FRAME
                                 * REFERENCE_CLOCK_FREQUENCY
```

```
                                        / SAMPLING_FREQUENCY);
    this.timestamp += TIMESTAMP_DELTA;

    RTPPacket.writeUInt32BE(this.timestamp, 4);
```

- el identificador de fuente SSRC, que será único, puesto que sólo se emplea una fuente en la aplicación, y aleatorio, porque se ha comentado que no hay previsión de que pueda colisionar con más fuentes. Se calcula en el constructor:

  ```
  this.ssrc = Math.floor(Math.random() * 100000);
  ```

  y se escribe en el paquete siempre con el mismo valor:

  ```
  RTPPacket.writeUInt32BE(this.ssrc, 8);
  ```

- añadir la cabecera específica para audio *MPEG*, donde ambos campos serán cero. El primero, MBZ, porque no se usa, y el segundo, *Fragment Offset*, porque el inicio del *frame* dentro de la carga de datos es inmediatamente al principio, puesto que no lleva más que eso:

  ```
  RTPPacket.writeUInt32BE(0, 12);
  ```

- completar el paquete añadiendo el *frame* MP3 a continuación de las cabeceras:

  ```
  payload.copy(RTPPacket, 16);
  ```

La última pieza de la aplicación es *UDPSender*, que maneja la comunicación con la red. Se encarga de enviar los paquetes que genera el protocolo RTP a la dirección *multicast* empleando UDP. Por tanto, se hará que responda a los eventos `'packet'` que genera aquél:

```
rtpserver.on('packet', function(packet){
  udpSender.broadcast(packet);
});   (Player.js)
```

Además, opcionalmente recaba información sobre la cantidad de paquetes y bytes enviados con propósitos estadísticos que cualquier cliente puede consultar enviándole un *datagrama*. El contenido del *datagrama* no importa, puede estar vacio. La respuesta UDP contendrá un objeto JSON serializado cuyos campos son `txPackets` y `txBytes`, por ejemplo:

```
stats = {
  txPackets : 0,
  txBytes : 0
};
```

Por defecto, no se responde a las peticiones de estadísticas con lo que habrá que habilitarlo una vez se haya creado la clase:

```
var udpsender = new UDPSender();
udpsender.enableStats(true);
```

Por esta doble función que desempeña entonces, esta clase consta de dos *sockets*:`txSocket`, para emitir al grupo de *multicast*, y `rxSocket`, para recibir las peticiones de los datos estadísticos:

```
var Sender = function(options){
  var options = options || {}
  this.port = options.port || 5000;
  this.broadcastAddress = options.broadcastAddress || '224.0.0.14';
  this.stats = {
      txPackets : 0,
      txBytes : 0
  };
  this.txSocket = dgram.createSocket('udp4');
  this.rxSocket = dgram.createSocket('udp4');
};
```

El primero de ellos se empleará en el método `UDPSender.broadcast(packet)` para transmitir un paquete cuando se reciba el evento `'packet'`. El envío se hace con el método que el *socket* proporciona:

```
var self = this;
this.txSocket.send(packet,
                   0,
                   packet.length,
                   this.port,
                   this.broadcastAddress,
                   function(err, bytes){
                     ++self.stats.txPackets;
                     self.stats.txBytes += bytes;
```

```
                    });
```

El último argumento es opcional, pero está presente porque es un *callback* que se ejecuta cuando Node termina de enviar los datos por el *socket*. Esto será útil para el propósito de mantener actualizadas las estadísticas, que será justamente en ese momento.

El segundo *socket*, `rxSocket`, escucha los mensajes que piden que se envíen las estadísticas al solicitante. En esta aplicación está asociado, arbitrariamente, al puerto 5001:

```
this.rxSocket.bind(5001);
```

Una vez habilitado para recibir, recibirá atendiendo el evento para ello, `'message'` enviando los datos solicitados de inmediato. A diferencia de `txSocket`, que transmite para una IP *multicast*, la respuesta debe ir dirigida al solicitante cuya IP no se conoce a priori pero que puede consultarse en el paquete de la petición. Estará contenida en la estructura `rinfo`, *remote info*, que el *listener* recibe junto con el mensaje:

```
var self = this;
if (enable){
  this.rxSocket.on('message', function(msg, rinfo){
    var stats = new Buffer(JSON.stringify(self.stats));
    dgram.createSocket('udp4').send(stats,
                                    0,
                                    stats.length,
                                    2468,
                                    rinfo.address);
  })
else{
  this.rxSocket.removeAllListeners();
}
```

Toda la lógica que establece la relación entre estos módulos está en el *script app.js*, y se ha ido introduciendo conforme se ha ido desgranando la interacción de los mismos.

3.6. Objetivos de los Koans

Al término de la realización de la aplicación se deberán conocer los aspectos más elementales de los módulos *buffer* y *dgram* con los que ser capaces de crear sencillas aplicaciones de red sobre el protocolo UDP.

De *buffer* se pretenden conocer las distintas funciones para escribir datos de diferentes tamaños en un *Buffer*.

1. Datos de 1 byte donde no se especifica *endianness* con `writeUInt8()`.

   ```
   RTPPacket.___(128, 0);
   ```

2. Datos de 2 bytes [BE] distinguiendo su endianness

   ```
   RTPPacket.___(this.seqNum, 2);
   ```

3. Datos de 4 bytes distinguiendo su endianness

   ```
   RTPPacket.___(this.timestamp, 4);
   ```

De *dgram* se pretende conocer el ciclo de vida de un *socket* UDP, con las primitivas más comunes que soporta para una comunicación completa.

1. Crear sockets con la función `dgram.createSocket()` sabiendo que hay que especificar si se va a trabajar sobre IPv4 o Ipv6:

   ```
   this.txSocket = dgram.___('udp4');
   this.rxSocket = dgram.___('udp4');
   ```

2. Ser capaz de vincular un *socket* a un puerto determinado en una interfaz concreto con `dgram.bind()`:

   ```
   this.rxSocket.___(7531);
   ```

3. Enviar un datagrama con `dgram.send()` y conocer qué argumentos acepta esta función:

   ```
   this.txSocket.___(packet,
                     0,
                     packet.length,
                     this.port,
                     this.broadcastAddress,
   ```

```
function(err, bytes){
  ++self.stats.txPackets;
  self.stats.txBytes += bytes;
});
```

4. Saber que hay que atender el evento `'message'` para recibir correctamente *datagramas* de clientes:

```
this.rxSocket.on(___, function(msg, rinfo){
  // ...
};)
```

5. Conocer la estructura `rinfo` que Node pone a disposición del programador con información del origen del *datagrama*:

```
function(msg, rinfo){
  var stats = new Buffer(JSON.stringify(self.stats));
  dgram.createSocket('udp4').send(stats,
                                  0,
                                  stats.length,
                                  5002,
                                  ___.address);
};
```

3.7. Preparación del entorno y ejecución de los Koans

Con objeto de poder ejecutar los Koans satisfactoriamente, se emplea un módulo, *koanizer*, que se distribuye con la práctica y cuya misión es adaptar el código para que puedan insertarse los huecos que caracterizan a un *koan*.

Los *koans* están distribuidos en dos ficheros: `dgram-koans.js`, que contiene los relacionados con el módulo *dgram*, y `buffer-koans.js`, que comprende aquellos que tienen que ver con el módulo *buffer*. Ambos deben ser editados y completados, sin que quede ningún hueco ___ (tres guiones bajos seguidos).

Verificar que se han realizado con éxito los *koans* se determina ejecutando los casos de prueba con:

```
$ jasmine-node -verbose spec/
```

Como resultado de una correcta resolución de los *koans*, la práctica es completamente funcional y se puede ejecutar con el comando:

```
$ node app.js
```

Para escuchar la música que se distribuye por RTP, hay que configurar un reproductor tal y como se indica en el apartado “Descripción de la aplicación”.

Si además se quiere comprobar la característica adicional que se ha introducido, *consulta de estadísticas*, puede hacerse a través del comando *netcat*[2]. Con *netcat* se pueden realizar operaciones de intercambio de datos en redes basadas en TCP/IP. En este caso, se empleará dos veces, para:

1. escuchar en un puerto los paquetes con información estadística que envía, bajo demanda, la aplicación. Se ha establecido, de manera coherentemente aleatoria, que el puerto donde se reciba sea 5002.

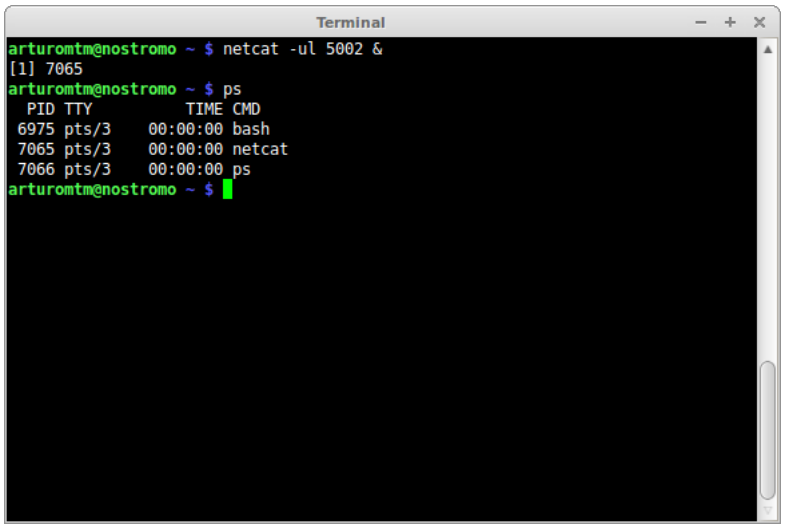

Figura 3.5: Ejecución de Netcat

2. solicitar dicha información, con un mensaje UDP, de contenido cualquiera, al puerto 5001 donde la aplicación los escucha.

[2] http://netcat.sourceforge.net/

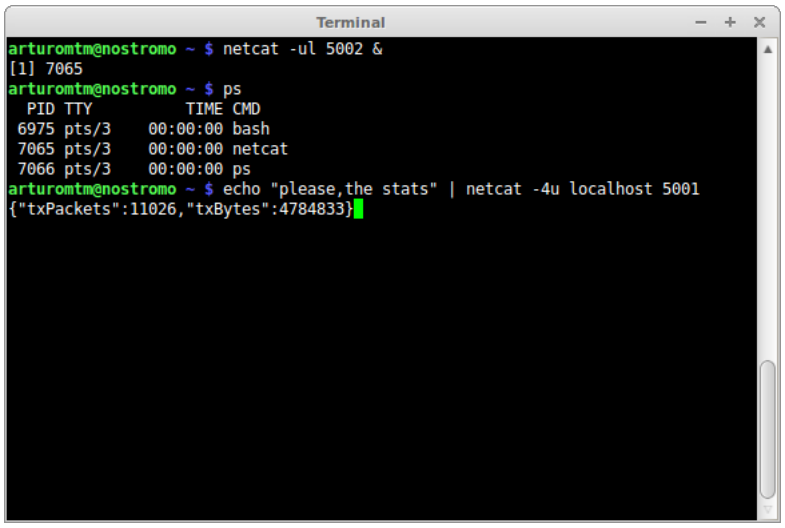

Figura 3.6: Respuesta a la petición con Netcat

3.8. Conclusión

Con el primer capítulo se ha introducido el desarrollo de una sencilla aplicación de Internet al estilo de Node donde se presentan sus características fundamentales. Éstas serán las que marquen la línea a seguir en este tipo de desarrollos, por ejemplo, poner a escuchar un programa en un puerto con `listen()`, realizar operaciones de Salida, con `send()`, o Entrada, atendiendo el evento `'message'`, y procesar los datos con los métodos de *Buffer*, como `write()`.

Capítulo 4

Módulos Stream y Net

4.1. Aspectos de TCP relevantes para Node

El protocolo de nivel de transporte TCP (*Transmission Control Protocol*) es, junto con IP, el más conocido de los protocolos de Internet precisamente por dar nombre a la pila de protocolos en la que se basan la inmensa mayoría de las aplicaciones de esta red.

TCP es un protocolo complejo, cuyas características son todo lo contrario a las de UDP. Comenzando con que TCP está orientado a conexión, es decir, se establece una conexión lógica punto a punto sobre la que los datos que se transmiten pueden ser vistos como un flujo bidireccional, o *stream full-duplex*. En un *stream* [46, Chapter 6.5.2], es invisible la manera en que se han fragmentado los datos para su envío, viéndose éstos a nivel de aplicación como un "chorro" continuo que se puede procesar conforme se recibe, sin que haya llegado al final, en contraposición a UDP donde se tiene la noción de datagrama y se tiene presente que los datos llegan "partidos" en esos datagramas hasta el nivel de aplicación. Se denominan *full-duplex* a las conexiones con bidireccionalidad lectura/escritura simultánea: se puede escribir y leer al mismo tiempo en ambos sentidos.

TCP es un protocolo fiable: los datos enviados se entregarán en destino sin errores y en el mismo orden en que se transmitieron. Para conseguir estas características, TCP dispone de mecanismos como los números de secuencia y los asentimientos, conocidos como ACK, a través de los cuales se indica qué octetos se han enviado y cuáles se han recibido. Hay algunos más, como el control de congestión,

que regula el flujo de datos entre emisor y receptor para evitar pérdidas, o el establecimiento de conexión, que inicializa los estados del protocolo. Todos son posibles gracias a los campos de la cabecera específicamente reservados con ese objetivo.

Como es lógico, toda esta evolución con respecto al protocolo UDP se ve reflejada en la cabecera de un segmento TCP. Aparte de los ya conocidos campos Puerto Origen, Puerto Destino y Checksum, aparecen varios más relacionados con los mecanismos anteriormente mencionados:

Número de Secuencia y Número de asentimiento

El número de secuencia es el índice que el primer octeto de datos que se está enviando, tiene en el buffer donde están todos los datos a enviar. Su utilidad en el receptor es la de ordenar los datos del segmento donde corresponda cuando llegue.

Por su parte, el número de asentimiento lo envía el receptor al recibir un segmento de datos. Contiene el número de secuencia que espera recibir en el siguiente segmento enviado por el emisor, o lo que es lo mismo, que hasta ese número, todos los octetos han sido recibidos.

El orden que siguen los números de secuenciaasentimiento en los sucesivos envíosrespuestas puede complicarse dependiendo de si se pierden paquetes y se necesita por tanto un reenvío, o llegan duplicados, etc. Es algo que queda fuera del estudio de este tema puesto que es transparente a la aplicación y no se puede controlar con el API de Node.

Flags

Para distintos estados y acciones del protocolo que activados indican que

URG el segmento contiene información urgente, priorizada respecto al resto de los datos.

ACK el segmento TCP está siendo utilizado para realizar el asentimiento del correspondiente segmento recibido y por tanto el número de asentimiento que contiene es válido.

PSH los datos deben ser enviados al receptor inmediatamente. Esto es porque TCP, si lo estima necesario, puede almacenar los datos que la aplicación le entrega hasta que considere que se pueden enviarlo a través del canal, es decir, el envío de datos puede no ser inmediato. Este flag,

push, activado en un segmento, hará que todos los datos se envíen.

RST la conexión debe ser reiniciada porque se ha detectado algún error en el transcurso de la comunicación TCP. Se pueden producir múltiples tipos de errores a lo largo de todos los estados de una conexión, como por ejemplo, la recepción de un segmento de asentimiento de datos que aún no se han enviado.

SYN el segmento TCP está intentando establecer una conexión y pretende sincronizarse.

FIN el emisor no va a enviar más datos y la conexión se puede liberar.

Tamaño de ventana
como parte del mecanismo de control de congestión, este campo indica el número de octetos que el receptor admite a partir del último que ha recibido correctamente (indicado por el número de asentimiento). [47, Chapter 2.6].

El resto del paquete básicamente lo componen campos opcionales, y la parte de datos, que también es opcional. En TCP, los segmentos no sólo se usan para la transmisión de información sino que también los hay que están compuestos por la cabeceras IP más TCP. Suelen ser segmentos de señalización como los que negocian el inicio y fin de la conexión, o los de asentimiento, en caso de que el flujo de datos sea principalmente unidireccional.

A lo largo de una comunicación sobre TCP entre un cliente y un servidor se pasa por diversas fases:

- Establecimiento de conexión una conexión se establece en tres pasos, a través de un proceso conocido como *three-way handshake*. Es aquí donde se negocian los números de secuencia y asentimiento de ambas partes. En el caso más simple y visto desde la parte que inicia el *handshake* ocurre lo siguiente:

  1. Se envía un segmento con el flag SYN activado y un número de secuencia inicial.

  2. Se recibe un segmento de asentimiento al anterior con los flags SYN y ACK activados. SYN indica que se la otra parte envía un número de secuencia válido y ACK, que acepta el número de secuencia que se le ha proporcionado en el paso anterior.

3. Se envía otro segmento de asentimiento con, únicamente, el flag ACK activado, con el que se acepta el número de secuencia recibido en el paso anterior.

Cualquier variación del procedimiento que se salga de la lógica propuesta desembocará en el envío o recepción de un segmento con el flag RST activado, con el consecuente reinicio de la conexión.

- Intercambio de datos una vez establecida la conexión se envían los datos en paquetes TCP. Durante el intercambio entran en juego varios mecanismos de transferencia de datos

  1. Transmisión fiable: intercambio de segmentos con números de secuencia y asentimiento para que todos se reciban en perfecto orden y sin duplicidad. Si se produjese la pérdida de algún segmento, se activa el mecanismo de retransmisión. Un segmento se supone perdido si no llega su asentimiento dentro de un intervalo de tiempo que el emisor calcula (*retransmission timeout*). Una vez vencido el *timeout*, el segmento se retransmite.

  2. Control de congestión: se realiza un ajuste dinámico del tamaño de la cantidad de datos que un extremo acepta y que notifica en la cabecera TCP mediante el campo "Tamaño de ventana". El ajuste de la ventana se realiza a través de algoritmos que implementa TCP y están fuera de lo que se pretende con el tema que nos ocupa. Sin embargo, hay un comportamiento interesante de TCP relacionado con el control de congestión: el algoritmo de *Nagle* [46, Chapter 6.5.8]. A grandes rasgos este algoritmo almacena en el *buffer* de transmisión los datos a enviar cuando son de pequeño tamaño con objeto de evitar malgastar ancho de banda al enviar más cabecera que *payload* y al reducir el tiempo que se tarda en enviar datos y recibir su correspondiente asentimiento (latencia). Además evita llenar demasiado rápido la ventana del receptor. Como efecto colateral, podrían observarse retardos en la ejecución de la aplicación que se comunica con el servidor de tal manera que si son significativos, cabría la posibilidad de plantearse el desactivarlo.

  3. Keep-alive [48, Chapter 4.2.3.6]: esta característica del protocolo es opcional y, de hecho, no está en la RFC original sino en la RFC1122. Es un mecanismo, por defecto desactivado, para comprobar que una conexión

por la que no se han recibido datos o asentimientos en mucho tiempo está todavía activa. Para ello se envían periódicamente segmentos TCP sin datos o con un octeto aleatorio como carga y se espera como respuesta un ACK, siendo un RST si ha habido interrupción de la conexión. Estos envíos se realizan después de que haya pasado un intervalo de tiempo configurable, o por defecto no inferior a dos horas según la RFC, sin actividad en el *socket* desde la última recepción de un segmento TCP.

Se recomienda, en caso de que se use, que sea por la parte servidora hacia el cliente para detectar caídas o desconexiones accidentales por parte de éste y liberar los recursos que se le habían asignado.

- Fin de conexión se produce cuando uno de los dos extremos (o ambos a la vez) no tiene más datos que enviar (lo que no quita que no pueda seguir recibiéndolos, por lo menos hasta que el otro extremo cierre).

  Un extremo cierra su conexión enviando un segmento con el flag FIN activado, que será confirmado con su correspondiente ACK por la otra parte. A partir del envío de FIN no será posible enviar más datos, pero sí recibirlos hasta que llegue un paquete FIN desde el extremo remoto. En este momento, se confirma con un ACK y ambos extremos se cierran.

  Destaca de este procedimiento que hay un momento en que la comunicación está "medio abierta": no se puede enviar pero sí recibir. Por tanto, para cerrar una conexión *full-duplex* no basta con señalizarlo desde un extremo sino que el otro debe señalizarlo también (si no tiene más datos que enviar). Sin embargo, estrictamente hablando, se dice que una conexión está "medio abierta" (*half-open*) si, según la RFC793, uno de los extremos se ha cerrado sin notificarlo al otro. En este caso se debería producir un reinicio de la conexión, RST [47, Chapter 3.4].

4.2. Streams en Node

Un *Stream* en Node 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 esos 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 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.

Se sabe que de un *Stream* se puede leer porque su propiedad `stream.readable` es `true`. La lectura se realiza de manera asíncrona, instalando un *Listener* para el evento `'data'`: `stream.on('data', function(datos){})`. Los datos que recibe la función de *callback* son por defecto de tipo *Buffer*, a menos que se especifique una codificación mediante `stream.setEncoding(codificacion)`.

El ciclo de vida de un *Stream readable* se puede gestionar de manera sencilla con una serie de métodos:

`stream.pause()`
: detiene la entrega de datos, es decir, no se emitirán eventos `'data'` con lo que la lectura quedará pausada. El API de Node advierte que pueden seguir emitiéndose algunos eventos `'data'`, cuya información se puede el programa puede almacenar, después de invocar al método hasta que la parte de Entrada/Salida, *libuv*, de Node procese la señal de parada.

`stream.resume()`
: reanuda la entrega de datos pausada con el método anterior

`stream.destroy()`
: anula toda posibilidad de operaciones Entrada/Salida sobre el *socket*. En consecuencia, no se emitirán más eventos `'data'` o `'end'` aunque sí cabe esperar un evento `'close'` cuando se liberen todos los recursos asociados al *Stream*.

Por su parte, se puede escribir en un *Stream* si su propiedad `stream.writable` es `true`, y se hace con el método `stream.write()` que adopta dos formas dependiendo de qué tipo es la información a escribir:

- si son datos en el formato por defecto, esto es, los bytes en bruto de un *Buffer*, se utiliza `stream.write(buffer)`.
- si se escriben datos codificados como texto (por defecto en `utf8` pero se puede cambiar opcionalmente con el argumento `codificacion`) se empleará `stream.write(string, [codificacion])`.

El ciclo de vida de este *Stream writable* se gestiona con un método que adopta varias formas:

`stream.end()`
: que finaliza el stream según corresponda a la naturaleza del *Stream*. Por ejemplo, si se trata de un fichero, escribe el carácter de control *EOF* (*End Of File*).

  Tiene una variante, que permite escribir datos en el stream antes de cerrar y luego, cerrarlo. Si los datos son bytes se emplea `stream.end(buffer)`, y si es texto, `stream.end(string, [codificacion])`.

`stream.destroy()`
: tiene idéntico efecto al del caso del *Stream readable*: impide que se realicen operaciones de Entrada/Salida, sin importar que queden datos por escribir.

  Si quedan datos por escribir y se desea que se escriban antes de que se cierre, el API ofrece el método `stream.destroySoon()` que también destruye el *Stream* directamente si no hay datos en la cola de escritura.

Para que una clase creada por el programador se comporte como un *Stream* se debe importar la clase base *Stream* contenida en el módulo *stream*, heredar de ella e implementar los métodos para las operaciones de lectura/escritura y gestión del *Stream*:

```
var Stream = require('stream'),
  utils = require('utils');

function myNewStream(){
  Stream.call(this);
  // codigo que implementa el constructor
}

utils.inherit(myNewStream, Stream);
```

```
myNewStream.prototype.write = function(datos, codificacion){
  // implementacion de la funcion
}
```

La clase base *Stream* únicamente ofrece el método `Stream.pipe()`:

`Stream.pipe(destino, [opciones])`

permite conectar un *Stream readable*, que será una fuente de datos binarios, a un `destino` *writable* donde se escriben esos datos de manera sincronizada. Esto lo consigue pausando y reanudando el *Stream* emisor según los datos se van escribiendo o no en el *Stream* `destino`. Cuando el emisor finaliza con el evento `'end'`, invoca al método `destino.end()` a menos que en las opciones se indique lo contrario:

```
opciones = { end: false };
```

La función devuelve el *Stream* `destino` con el propósito de que puedan encadenarse varios *Streams* consecutivamente. Por ejemplo, se conectan entre sí las entradas y las salidas de los *Streams* `A`, `B` y `C` de la manera:

```
A.pipe(B).pipe(C) // equivale a A.pipe(B); B.pipe(C);
```

4.3. TCP en Node

Una aplicación Node puede hacer uso de TCP importando la librería *net*:

```
var net = require("net");
```

net proporciona al programador el objeto *Socket*, que representa obviamente un *socket* TCP o un *socket* de dominio UNIX[1], 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. Sin embargo, ninguno de los dos objetos

[1]Los *Unix Domain Sockets* son un mecanismo de comunicación interproceso *IPC* como lo son, por ejemplo, las *pipes* (tuberías). A diferencia de éstas, los *Unix Domain Sockets* son un mecanismo bidireccional y además, las operaciones no se hacen a través de las funciones para el manejo de ficheros sino a través de las que se emplean con los *sockets*. Por tanto, no quedan descritos por un descriptor de fichero, ni tampoco por una dirección IP sino por una ruta a un archivo del sistema de ficheros.

se instancia directamente: se debe hacer a través de métodos específicos para ellos.

Desde la parte cliente, obtenemos un *socket* creando una conexión a una ubicación concreta mediante los métodos `net.connect()` o `net.createConnection()` indistintamente. Ambos métodos tienen varias cabeceras que aceptan distintos argumentos. Las más generales son:

`net.connect(opciones, [function(){ }])`
`net.createConnection(opciones, [function(){ }])`

El argumento `opciones` contiene la información necesaria para establecer la conexión. En el caso de conexiones de red TCP esto incluye el puerto `port` (única opción obligatoria), la máquina destino `host` y la interfaz local `localInterface` desde el que realizar la conexión. Por ejemplo:

```
{ port: 5479,
  host: "192.168.1.32",
  localInterface: "192.168.1.26" }
```

En el caso de *sockets* de dominio UNIX, la única opción, obligatoria, es la ruta *path* en el sistema de ficheros de la máquina donde corre el script.

Para ambos casos existe una opción común, `allowHalfOpen`, por defecto a `false` pero que si se activa permite mantener la conexión "medio abierta" cuando el otro extremo cierra su parte, dando al programador la posibilidad de gestionar él mismo qué hacer. A nivel de protocolo, esto significa que cuando se recibe desde el otro extremo un segmento FIN, no se envía otro FIN, como se ha visto que se comporta el protocolo en la fase de Fin de Conexión, sino que el programador puede seguir enviando datos (recordar que *half-open* implica que sí se puede recibir) o cerrar la conexión explícitamente con `socket.end()`.

El resto de cabeceras son una simplificación de las anteriores:

`net.connect(puerto, [maquina], [function(){ }])`
`net.createConnection(puerto, [maquina], [function(){ }])`

se emplean para conexiones TCP exclusivamente.

`net.connect(path, [function(){ }])`
`net.createConnection(path, [function(){ }])`

son las cabeceras para emplear *sockets* de dominio UNIX.

Todos los métodos emiten el evento `'connect'` de tal manera que instalar un *listener* para ese evento con `socket.on('connect', function() )` es equivalente a invocar a los métodos pasándoles un *callback*.

Cuando a través de estos métodos se ha obtenido una conexión, es decir, una instancia de *Socket*, se dispone de un flujo de datos sobre el que leer y escribir. Este *socket* se puede configurar con una serie de métodos accesores:

`socket.setTimeout(timeout, [function(){ }])`
: activa en el *socket* un temporizador, desactivado por defecto, que expira tras `timeout` milisegundos de inactividad, emitiendo el evento `'timeout'`. Este evento es capturable con el *callback* opcional del segundo argumento o a través de `socket.on('timeout', function() )`.

  El *timeout* no implica que expire la conexión o que se modifique de alguna manera. Si se quiere cerrar hay que llamar a los métodos existentes para ello, `socket.end()` o `socket.destroy()`, detallados más adelante.

  Para desactivarlo de nuevo, se invoca la función con el argumento `timeout = 0`.

`socket.setNoDelay([sinRetardo])`
: activa o desactiva, dependiendo del valor del argumento *booleano* `sinRetardo`, el algoritmo de *Nagle*. `sinRetardo` por defecto es `true`, con lo que los datos se envían de inmediato por el *socket* y, por tanto, indica que el algoritmo está desactivado.

`socket.setKeepAlive([activo], [retardoInicial])`
: activa o desactiva, con el argumento *booleano* `activo`, el mecanismo de *Keep-Alive*, por defecto desactivado. El `retardoInicial` indica en milisegundos el tiempo que debe estar el *socket* inactivo para que entre en acción.

  Si el valor de `retardoInicial` es cero, se toma como *timeout* el anterior valor de `retardoInicial` o, de no ser posible, el valor por defecto, que es el que especifica la RFC1122 y que implementan las pilas TCP de los sistemas operativos (no Node): 2 horas (7200000 milisegundos).

`socket.setEncoding()`
: independiente del protocolo TCP en sí (no se modifican parámetros relativos

a la conexión) y orientado a la codificación de los datos que transporta para su consumo en el programa. Por defecto, la codificación es la estándar de Node, `utf8`.

Una vez conectado, además, se conocerán los datos de la conexión a través de:

`socket.address()`
: proporciona la dirección IP address de la interfaz donde está establecido el *socket*, la familia `family` de dicha dirección (IPv4 o IPv6) y el puerto local `port` al que está ligado, todo encapsulado en un objeto JSON, como por ejemplo

```
{ address: "192.168.1.35",
  port: 5556,
  family: "IPv4" }
```

`socket.remoteAddress`
: es la dirección IP del otro extremo del *socket*

`socket.remotePort`
: es el puerto que emplea el otro extremo del *socket*

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*, a través de los métodos habituales de este tipo de objetos, comentados en el apartado anterior.

Para escribir datos en una codificación concreta en un *socket* se emplea:

`socket.write(data, [codificación], function(){ })`
: donde la función de *callback* se ejecuta cuando los datos han salido del *buffer* para ser enviados a través del *socket*.

Por su parte, la lectura se realiza como en cualquier *Stream*, atendiendo al evento `'data'`:

```
socket.on('data', function(data){
   //codigo para procesar los datos
});
```

Se puede además gestionar su ciclo de vida con los métodos habituales para ello, tanto los de los *Streams writable* como los de los *readable*:

`socket.pause()`
: el API de Node lo recomienda junto con `resume()` para ajustar la tasa de transferencia en la recepción de datos (por parte del servidor en los provenientes del cliente) ya que, como se ha explicado, `pause()` detiene la recepción de datos.

`socket.resume()`
: reanuda la recepción de datos detenida con el método `pause()`. De esta manera se vuelven a emitir eventos de tipo `'data'`.

`socket.end([datos], [codificacion])`
: si se invoca sin argumentos, comienza la fase de final de conexión enviando un paquete FIN, quedando ésta "medio cerrada": no se podrá escribir en el *socket* pero sí se podrán recibir datos del otro extremo (si este extremo no ha finalizado su conexión también con `end()`).

  En caso de que se invoque con argumentos, primero se escriben los `datos` con la codificacion especificada, como con `socket.write(datos, codificacion)`, y luego procede como si se invocase a `socket.end()`.

`socket.destroy()`
: inhabilita completamente las operaciones de Entrada/Salida en el *socket*, por lo que se usa en los casos en que se haya producido un error.

En la parte servidor, un *socket* es capaz de vincularse a un puerto sobre el que escuchar conexiones de otras máquinas e interactuar con ellas. Para este propósito el módulo net ofrece el método

`net.createServer([opciones], [function(socket){ }])`
: devuelve una instancia de la clase *Server* configurado con unas opciones que, hasta la fecha, sólo permiten mantener una conexión medio abierta con `allowHalfOpen` (por defecto `false`), que funciona como ya se ha explicado.

  ```
  opciones = { allowHalfOpen: true };
  ```

  El segundo argumento es un *listener* del evento `'connection'`, que se emite cada vez que hay una nueva conexión en el servidor. Esta función recibe como parámetro el *socket* que se crea. Siempre se puede atender a

`'connection'` como con todos los eventos de las instancias de la clase *EventEmitter*: `server.on('connection', function(socket){ })`.

La realidad es que una llamada a `net.createServer()` es equivalente a instanciar la clase *Server* directamente:

```
new Server([opciones], [function(socket){ }]);
```

El resultado del método es un servidor preparado para escuchar en distintos dispositivos según se emplee el método `listen()`:

`server.listen(puerto, [maquina], [backlog], [function(){ }])`

`server.listen(path, [function(){ }])`

`server.listen(handle, [function(){ }])`

Una vez levantado el servidor, se tiene disponible información relativa al mismo con:

`server.address()`
: que funciona de manera idéntica a la función homónima de la clase *Socket*, devolviendo un objeto JSON con tres propiedades: dirección IP `address`, tipo de IP `family` (v4 o v6) y puerto de escucha `port`. Por ejemplo:

```
{ port: 6420,
  family: 'IPv4',
  address: '127.0.0.1' }
```

`server.connections`
: es la propiedad que lleva la cuenta del número de conexiones concurrentes que maneja el servidor en ese momento

`server.maxConnections`
: es la propiedad que fija el número máximo de conexiones concurrentes en el servidor, es decir, es el límite superior de la anterior propiedad. Cuando se alcanza este número, las conexiones se rechazan, abortando el intento de conexión en la parte cliente.

La gestión del ciclo de vida de un *socket* servidor es sencilla:

- a través del evento `'connection'`, `server.on('connection', function(socket){})`, se manejan las conexiones entrantes. La función de

callback recibe el *socket* correspondiente a la conexión, el cual, obviamente, es una instancia de *Socket* y, por tanto, se maneja como tal.

- `server.close([function(){ }])` impide que el servidor admita nuevas conexiones, pero no lo destruye hasta que no se hayan cerrado todas. Será entonces cuando se emita el evento `'close'` que se puede atender con un *callback* pasándolo como argumento opcional o instalándolo como *Listener* con `server.on('close', callback)`.

Durante la ejecución del servidor pueden producirse errores que se procesan capturando el evento `'error'`:

```
server.on('error', function(error){ })
```

4.4. Aplicación con Streams TCP

Una aplicación ejemplo que puede implementarse con TCP es la de un servidor que acepte conexiones a través de las cuales reciba comandos que modifiquen su comportamiento. La salida del comando será enviada de vuelta al cliente.

4.4.1. Descripción del problema

En esta práctica se realizará una modificación de la aplicación UDP: se adaptará el emisor RTP para que envíe paquetes RTP a la dirección IP de un cliente en lugar de hacerlo a la IP de un grupo *multicast*. Para tal propósito, el cliente debe conectarse a un puerto del servidor y, a través de comandos introducidos a través de un *prompt*, manejar la lista de reproducción del servidor. Se definirán seis comandos posibles:

- list: ofrece una lista de canciones disponibles para escuchar
- play: inicia la reproducción de la lista, por defecto parada
- pause: detiene la reproducción en un punto concreto de una canción
- next: solicita la reproducción de la siguiente canción
- prev: solicita la reproducción de la canción anterior
- exit: finaliza la sesión

Puesto la aplicación se basa en un intérprete de comandos, seguirá siendo necesario un reproductor de audio que soporte RTP que apunte a la dirección IP de la máquina desde donde se realice la conexión a la aplicación.

4.4.2. Diseño propuesto

Los requisitos de esta aplicación hacen necesaria una modificación del diagrama de módulos respecto al que modelaba la aplicación UDP/RTP. En este escenario se introduce la clase *RemotePrompt* en el módulo principal de la aplicación y será donde se implementen las características que se le han pedido al servidor. El diagrama que modela el escenario para esta práctica sería el siguiente:

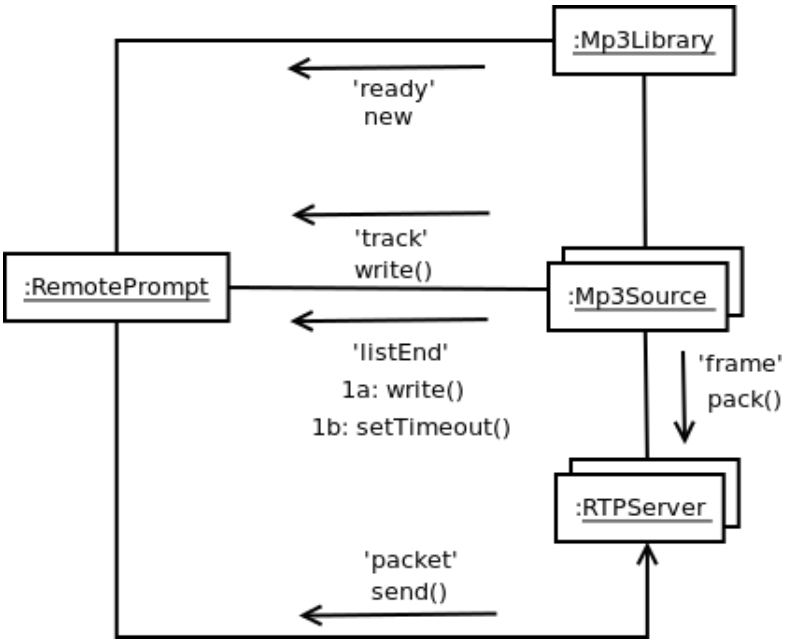

Figura 4.1: Diagrama de Colaboración de la Solución TCP

El funcionamiento general de la aplicación es, a grandes rasgos, escuchar y atender las conexiones que se produzcan en el puerto que elija quien emplee el módulo (por ejemplo, 2323), rechazando aquellas que ya tengan una sesión establecida. Para conocer este dato, se mantiene una tabla de direcciones IP remotas, `sessionsDB`:

```
var sessionsDB = {};
this.server = net.createServer();
```

```
this.server.on('connection', function(connection){
  var remoteIP = connection.remoteAddress;

  connection.write("Welcome to your command line playlist manager, "
                   + remoteIP);

  if (remoteIP in sessionsDB){
    connection.end("Duplicated session, closing.");
    return;
  };

  sessionsDB[remoteIP] = true;

  // Logica de la aplicacion referente
  // al tratamiento de la conexion
});
```

Por cada una de las conexiones anteriores se creará una fuente de audio MP3, *MP3Source*, que controle el estado de la reproducción. Esta clase proporcionará para ese propósito unas funciones que se invocarán según demanden los comandos. Además, informará a través de eventos sobre aspectos relevantes de la reproducción que son útiles que el cliente conozca, notificándoselo automáticamente. En concreto, mediante el evento `'track'`, *MP3Source* avisará de cuándo se ha realizado una operación sobre una canción, bien bajo demanda del cliente o bien porque sea inherente al transcurso de la reproducción, como cuando se pasa a la siguiente pista:

```
source.on('track', function(trackName){
  connection.write("Now playing " + trackName + "\r\n# ");
});
```

Mediante el otro evento, `'listEnd'`, se informa cuándo se ha alcanzado el final de la lista de reproducción. Éste además activa el mecanismo de cierre automático de la aplicación tras un intervalo de tiempo sin actividad:

```
source.on('listEnd', function(){
  var seconds = 10;
  connection.write("End of the list reached.Closing in "
                   + seconds
```

```
                    + " seconds\r\n# ");
  connection.setTimeout(seconds * 1000, function(){
    delete sessionsDB[this.remoteAddress];
    connection.end("Your session has expired. Closing.");
  });
});
```

La transmisión de los paquetes RTP se delega en la clase *RTPServer* que, como en la práctica anterior sobre UDP, está íntimamente ligada con la clase *MP3Source*. Para esta práctica se ha simplificado al máximo este aspecto de la conexión, quedando simplemente en el trozo de código siguiente, sin mayor interés para el objeto de la práctica:

```
var rtpserver = new RTPServer();
var udpSocket = udp.createSocket('udp4');

rtpserver.on('packet', function(packet){
  udpSocket.send(packet, 0, packet.length, 5002, remoteIP);
});

source.on('frame', function(frame){
  rtpserver.pack(frame);
});
```

Los comandos se reciben a través del *socket* de la conexión. Se ha visto que para recibirlos, y recibir cualquier tipo de datos en general, es necesario atender al evento `'data'` para lo que habrá que instalar el correspondiente *listener*: será el encargado de procesar los comandos y actuar en consecuencia sobre la fuente MP3.

```
connection.on('data', function(data){
  this.setTimeout(0);
  // Logica del procesado de los comandos
});
```

En el código anterior se puede observar que, como medida preventiva, se desactiva el *timeout* con el que se cierra el *socket*, por si hubiera sido activado.

A la hora de analizar los comandos, se debe tener en consideración que provienen del *buffer* de red con un retorno de carro que es conveniente limpiar para,

posteriormente, proceder a identificarlos:

```
var command = data.toString('utf8').split("\r\n")[0];
```

Por lo general, cada comando identifica un método homónimo de la fuente de audio y la mayoría de las veces, no supone más que esa operación. En otros casos, como el del comando "list", se formatea la salida para presentarla al cliente de una manera visualmente ordenada:

```
switch(command){
  case "list":
    var playlist = source.list();
    this.write("\r\nSongs in the playlist");
    this.write("\r\n----------------------");
    for (var i=0; i < playlist.length; i++){
      var song = playlist[i];
      this.write("\r\n"
                 + (source.currentTrack() == song? "> " : "  ")
                 + song);
    }
    this.write("\r\n# ");
    break;

  case "play":
    source.play();
    break;

  case "pause":
    source.pause();
    break;

  case "next":
    source.next();
    break;

  case "prev":
    source.prev();
    break;
```

```
  case "exit":
    delete sessionsDB[this.remoteAddress];
    this.end("Bye.");
    break;

  default:
    this.write("Command " + command + " unknown\r\n# ");
}
```

Exceptuando "exit", que no actúa sobre la fuente de audio, el resto de los comandos provocarán que *MP3Source* emita el evento `'track'`. Así, se ofrecerá información al cliente sobre el resultado de su acción, como se ha comentado durante el desarrollo de la solución propuesta.

Por último, sería necesario liberar los recursos ocupados por un cliente una vez éste haya finalizado su sesión ya sea porque la cierra el cliente (con "exit"), el servidor (por `'listEnd'`) o se corta la conexión. La mejor manera de conocer esto es atendiendo el evento `'close'` del *socket*:

```
connection.on('close', function(){
  source.pause();
  udpSocket.close();
  rtpserver = source = null;
});
```

Toda la lógica anterior, menos la parte que pone a escuchar al servidor, está contenida en el constructor del objeto *RemotePrompt*. Aparte del constructor, *RemotePrompt* también posee un método `listen()`, con el que comenzar la actividad de la aplicación, puesto que se encarga de ordenar al servidor comenzar a escuchar en un puerto:

```
this.listen = function(port){
  this.server.listen(port);
}
```

Sin embargo, a pesar de tener un constructor, *RemotePrompt* sólo es instanciable desde el módulo `RemotePrompt.js` a través del método `create()` que es lo único que éste exporta, no pudiendo hacerse de otra manera (por ejemplo con `new`). El motivo es que construir la librería de archivos *MP3Library*, otro de los cometidos

de `create()`, consume un tiempo indeterminado que debe esperarse y durante el cual, el objeto *RemotePrompt* está en un estado indefinido. Con `create()` se eliminan los errores derivados de esta espera, pues gestiona ella misma el evento `'ready'` de la librería, y ofrece una instancia de *RemotePrompt* funcional desde el primer momento. El siguiente fragmento de código explica lo anterior, aunque no sería necesario conocerlo para el desarrollo de la práctica:

```
exports.create = function(){
  var app;
  var library = new nodeMp3.Mp3Library({ basedir: '../data/songs/' });

  library.on('ready', function(){

    app = new RemotePrompt(this);
    // Gestion de metodos invocados de app
    // cuando estaba en proceso de creacion
  });
  // Logica de gestion del objeto
  // app para uso del desarrollardor
}
```

Así, ejecutar la aplicación puede hacerse desde un sencillo *script*:

```
var remotePrompt = require('./RemotePrompt');
var app = remotePrompt.create();
app.listen(2323);
```

o incluso a través de una sóla línea de código en la línea de comandos de Node:

```
> require('./RemotePrompt').create().listen(2323)
```

4.5. Objetivos de los Koans

A través de los Koans desarrollados para esta aplicación se busca obtener un conocimiento básico del servidor TCP que Node pone a disposición del programador y de las conexiones que genera, tanto de su manejo a través de sus métodos principales como de algunas propiedades destacables de ellas.

1. Emplear el método `createServer()` para disponer de un servidor para conexiones TCP:

   ```
   this.server = net.___();
   ```

2. Hacer que el servidor TCP escuche en un determinado puerto invocando su método `listen()`:

   ```
   this.server.___(port);
   ```

3. Atender cada una de las conexiones que acepta el servidor mediante la escucha del evento `'connection'` que él mismo emite:

   ```
   this.server.on(___, function(connection){});
   ```

4. Escribir en el *Stream* de la conexión con el método `write()`:

   ```
   connection.___("Welcome to your command line playlist manager, "
                  + remoteIP);
   ```

5. Finalizar una conexión a través del método `end()` de la misma:

   ```
   connection.___("Duplicated session, closing.");
   ```

6. Activar con `setTimeout()` el contador de *timeout* en el *socket* para realizar una acción concreta tras ese tiempo:

   ```
   connection.___(seconds * 1000, function(){
     delete sessionsDB[this.remoteAddress];
     connection.end("Your session has expired. Closing.");
   });
   ```

7. Recibir datos del cliente de manera asíncrona escuchando el evento `'data'` de la conexión:

   ```
   connection.on(___, function(data){  });
   ```

4.6. Preparación del entorno y ejecución de los Koans

De nuevo, un requisito para poder ejecutar los Koans satisfactoriamente es el módulo *koanizer*, distribuido con la práctica.

El fichero que contiene los *koans* es `net-koans.js`, que es en realidad el módulo *RemotePrompt*, puesto que es éste el que lleva todo el código asociado al módulo *net*. Al igual que en la práctica anterior debe ser editado y completado, sin dejar ningún hueco ___.

Para verificar que se han realizado con éxito los *koans* se ejecutan los casos de prueba con:

```
$ jasmine-node -verbose spec/
```

Como resultado de una correcta resolución de los *koans*, la práctica es completamente funcional y se puede ejecutar con el comando:

```
$ node app.js
```

Para escuchar la música que se distribuye por RTP, hay que configurar un reproductor tal y como se indicó en el apartado "Descripción de la aplicación" del anterior capítulo. Esta vez, sin embargo, los paquetes se reciben en la dirección IP de la interfaz de la máquina que se conecta al servidor. Cuando se realiza una conexión al servidor, éste informa de la dirección donde hay que apuntar el Reproductor. Para conectarse bastará un simple `telnet`:

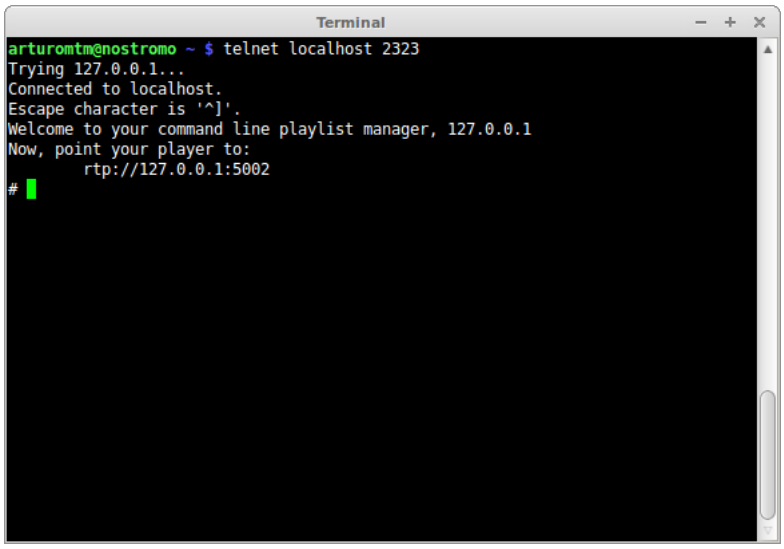

Figura 4.2: Telnet al puerto de la aplicación

Conocida ésta, se puede proceder a configurar el Reproductor:

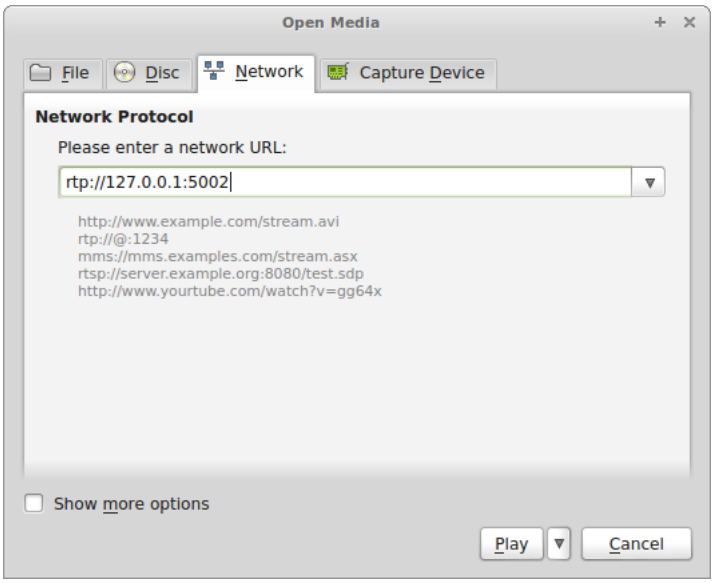

Figura 4.3: Reproductor VLC, configuración

Para comenzar la reproducción es necesario avisar al servidor con *"play"* en el *prompt* de la aplicación donde pueden también, introducirse el resto de comandos:

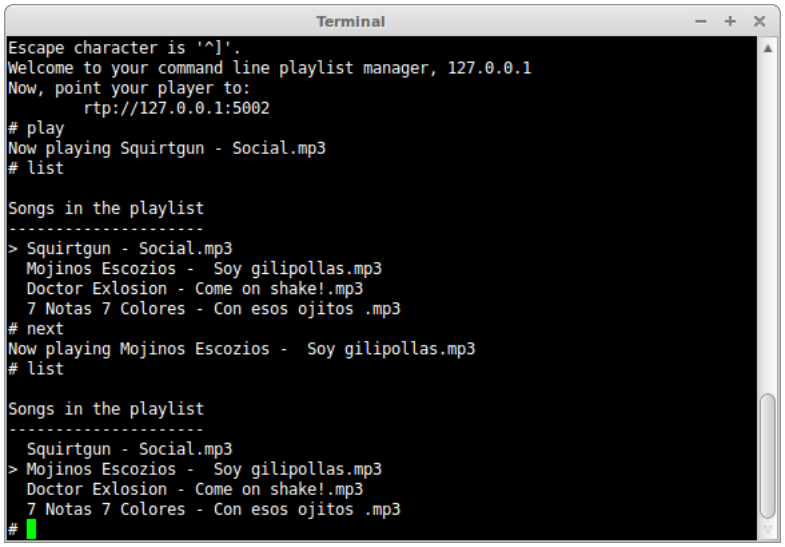

Figura 4.4: Solución TCP, salida de los comandos

4.7. Conclusión

Completando el problema propuesto como aplicación de este capítulo, se habrán adquirido los conocimientos necesarios para crear un servidor TCP mediante `createServer()` que admita conexiones, atendiento al evento `'connection'`, en un puerto (con `listen()`). Se debe ser capaz de realizar un intercambio de datos basándolo en la recepción con un *listener* del evento `'data'` y el envío, escribiendo en el *Stream* correspondiente con `write()`. Una vez que ambos extremos hayan completado la comunicación, se debe ser capaz de finalizarla con `end()`.

Capítulo 5

Módulo Http

5.1. Aspectos de HTTP relevantes para Node

HTTP (*Hyper Text Transfer Protocol*) es un protocolo de nivel de aplicación para el acceso a recursos en entornos distribuidos. Estos recursos están identificados unívocamente por su URL (*Uniform Resource Locator*) que es su localizador: indica la ruta para acceder al él. Las URLs son un subconjunto de los denominados URIs, y un URI (*Uniform Resource Identificator*) no es más que, como su nombre indica, un identificador del recurso. Por tanto, URL y URI son términos equivalentes en este caso: un recurso está identificado por su localización.

El formato más típico de una URL es:

```
<protocolo>://<maquinaremota>:<puerto>/<ruta>/<al>/<recurso>
[?<query>=<valor>]
```

Por el tema que nos ocupa, `<protocolo>` es obvio que es `http`, aunque podría ser `https` si la conexión sobre la que se establece la comunicación fuese un canal seguro. La `<maquinaremota>` es el nombre de DNS o la dirección IP de la máquina donde corre el servidor HTTP, general, pero no necesariamente, escuchando en el `<puerto>` 80 que es el asignado por la IANA[1] para HTTP. La `<ruta>/<al>/<recurso>` es el *path*, en general relativo en el sistema de ficheros, del servidor donde se aloja el recurso. Por último, en el caso de que el recurso sea

[1] *Internet Assigned Numbers Authority* es el organismo encargado de fijar de manera oficial los códigos y números que se emplean en los protocolos estándar de Internet como rangos de IPs o números de puerto

dinámico, es decir, sea la salida de un programa ejecutable localizado en la URL, se le pueden pasar parámetros de entrada en la parte opcional de la misma, la `<query>`. Una *Query* comienza en la URL con el signo de interrogación '?' a partir del cual se concatenan con el símblo & los parámetros de nombre `<query>` y su `<valor>` asociado con un =. Ilustrando todo lo anterior:

```
http://www.google.com/search?q=nodejs+koans
```

El escenario de uso del protocolo implica, en su manera más básica, a dos actores que interactúan mediante un modelo de intercambio de mensajes denominados petición, para la parte del cliente, y respuesta, para la del servidor. Este intercambio no tiene estado, es decir, los mensajes son independientes de anteriores y posteriores.

Los actores implicados son principalmente el cliente, a través del Agente de Usuario, que es aquel programa que genera las peticiones, y el Servidor, que las recibe y es capaz de interpretar el protocolo en el extremo opuesto.

El escenario básico puede complicarse con la aparición de elementos intermedios (programas o dispositivos) entre cliente y servidor, como *proxies*, *gateways* o túneles. Un *proxy* es un intermediario entre cliente y servidor, a menudo transparente para ambos. Por este motivo es capaz de comunicarse con los dos extremos actuando al mismo tiempo como cliente y como servidor. Su misión es variada pero principalmente tiene propósitos de seguridad y *caché*. La función de la *caché* es optimizar recursos de red, almacenando durante un tiempo los recursos a los que se accede a través suyo para entregarlos al cliente si los vuelve a solicitar y así no tener que pedirlos de nuevo al servidor, ahorrando el tiempo y ancho de banda que ello supone.

Un *gateway* es también un intermediario pero esta vez de la parte servidora. Su propósito es recibir las peticiones dirigidas al servidor, adaptarlas para reenviárselas, tanto a nivel de mensaje como a nivel de protocolo si procediera, y devolver las respuestas de vuelta al cliente.

Un túnel se puede considerar, como explica la RFC [49, Section 1.3], como un intermediario que actúa de enlace entre dos conexiones, sobre las que pueden ir distintos protocolos. El mecanismo de tunelado permite a la máquina origen, cliente, establecer una conexión punto a punto a través del túnel, que dejará de ser parte de la comunicación HTTP y se convertirá en un elemento transparente, dejando de existir cuando se cierre la comunicación en los extremos.

5.1.1. La parte del Cliente

Las peticiones HTTP están formadas por líneas de texto claro, terminadas con los caracteres de control <*CRLF*> (Retorno de carro más salto de línea). La primera línea, *request-line*, es siempre obligatoria, y el resto, las cabeceras y cuerpo, aportan información adicional. El esquema que sigue la *request-line* son tres campos separados entre ellos por un espacio:

<METODO> <URI> <VERSION>

El <METODO> es la acción que se solicita al servidor que realice sobre el recurso solicitado. Las distintas versiones del protocolo han ido introduciendo métodos, si bien, los únicos que todo servidor HTTP debe soportar obligatoriamente según la RFC son:

GET
: devuelve el recurso identificado por la <URI>. Puede ser estático o pueden ser datos generados dinámicamente.

HEAD
: devuelve información sobre el recurso solicitado siendo dicha información la misma que proporcionaría GET pero sin incluir el recurso en sí en la respuesta

Los dos métodos anteriores se denominan “Métodos seguros” [49, Section 9.1.1] puesto que la acción que se solicita con ellos únicamente tiene como resultado la obtención de un recurso del servidor y/o información sobre él. Hay otros métodos que se pueden considerar inseguros porque pueden producirse resultados potencialmente peligrosos:

POST
: solicita al servidor el procesado del cuerpo de la petición con objeto introducir nueva información en el recurso identificado por la URI como, por ejemplo y según la RFC2616, aportar datos desde un formulario o introducir información en una base de datos.

PUT
: solicita al servidor crear o modificar el recurso identificado por la URI.

DELETE
: solicita al servidor la eliminación del recurso identificado por la URI.

De todos los anteriores métodos, GET, HEAD, PUT y DELETE se pueden considerar "Métodos idempotentes", es decir, que los efectos colaterales de varias peticiones de uno de ellos son los mismos que si se invocase el método una sola vez.

Del resto de métodos se puede destacar CONNECT. Con él se insta a un *proxy* a establecer un túnel con el *endpoint* especificado en la URI. El comportamiento del *proxy* al recibir el método CONNECT será realizar una conexión TCP al *endpoint*, como si del cliente se tratara, y redirigir el tráfico de manera transparente entre cliente y servidor, sin necesitar HTTP más que para iniciar el mecanismo de tunelado. De esta manera se puede, entre otras cosas, alternar entre HTTP y HTTP Seguro cuando hay un *proxy* HTTP entre cliente y servidor.

A lo largo del tiempo el protocolo HTTP ha sufrido dos actualizaciones desde su versión original, la 0.9, ahora obsoleta y fuera de uso. La última es la 1.1 habiendo mejoras posteriores experimentales, como la *RFC2774: An HTTP Extension Framework*. En definitiva, el campo <VERSION> de la *request-line* se compone de la cadena HTTP seguida del número de versión. En el caso de la última sería `HTTP1.1`.

La versión 1.1 añade muchas mejoras respecto a la 1.0, entre las cuales se pueden destacar:

- Nuevos métodos: HTTP 1.0 contempla los métodos GET, HEAD y POST únicamente, el resto de métodos comentados anteriormente y alguno más, como OPTIONS, pertenecen a HTTP 1.1.
- Conexiones persistentes: con HTTP 1.0, cada petición se realizaba bajo una conexión establecida con tal propósito. Una vez recibida la respuesta, la conexión se cerraba. HTTP 1.1 permite conexiones persistentes: sobre una misma conexión TCP se pueden realizar las peticiones HTTP necesarias, con el consiguiente ahorro de recursos en las máquinas (cliente, servidores y elementos de red) y la reducción de la latencia y la congestión de red, entre otras cosas.
- Control de *caché*: se introducen mecanismos de control de *caché* como, por ejemplo, nuevas cabeceras condicionales que complementan a la existente *If-Modified-Since* de la versión 1.0.
- Nuevos códigos de respuesta: HTTP 1.0 tenía reservado el rango de los có-

digos *1xx*, que devuelve el servidor como información al cliente para casos que pudiesen surgir en un futuro, pero no tenía definido ninguno. HTTP 1.1 introduce dos: `100 Continue` y `101 Switching Protocols`.

En el resto de rangos introduce algunos más y además desambigua otros existentes, como `302 Found` con los nuevos `303 See Other` y `307 Temporary Redirect`.

La petición puede constar de campos de cabecera adicionales, explicados al detalle en la RFC, con los que añadir información útil para el servidor a la hora de componer una respuesta. Con ellos se puede aportar, entre otros muchos datos:

- el tipo MIME de los recursos que acepta el Agente de Usuario, mediante la cabecera `Accept`. Un tipo MIME clasifica el formato de un recurso según su naturaleza: texto, imagen, audio...

  Los tipos MIME están definidos en la RFC1341, expresan los formatos de la entidad que, en este caso, acepta el Agente de Usuario y definen varios tipos y subtipos para clasificar el contenido binario del cuerpo de un mensaje según su naturaleza [41]. El formato MIME se expresará entonces como la dupla tipo/subtipo. Ejemplos de tipos MIME [46] con algún subtipo de los más utilizados son:

Contenido	Tipo MIME (*tipo/subtipo*)	Descripción
Texto	text/plain	Sólo texto
	text/html	Lenguaje de marcado hipertexto para páginas web
Imagenes	image/gif	Formato de intercambio de gráficos *gif*
	image/jpeg	Imágenes codificadas según el estándar propuesto por el *Joint Photographic Experts Groups*
Audio	audio/mpeg3	Audio en formato *MP3* según el estándar *MPEG-1*
Vídeo	video/mpeg	Vídeo en el estándar de codificación del *Moving Picture Experts Group*
Datos	application/octet-stream	Secuencia binaria genérica

Tabla 5.1: Ejemplos de tipos MIME

- qué lenguaje es el preferido y que codificación adicional y juego de caracteres desea el cliente que se emplee: `Accept-Language`, `Accept-Encoding` y `Accept-Charset` respectivamente.
- la referencia al recurso, identificado por su URI, del que se ha obtenido la URI de la petición, es decir, de dónde se proviene: `Referer`.
- el nombre de dominio y puerto de la máquina tal y como aparece en la URL: `Host`. Cuando una misma dirección IP tiene asignados en DNS varios nombres de dominio (*hosting virtual*), esta cabecera obligatoria permite localizar el recurso que se solicita en la URI.
- la versión 1.1 del protocolo incluye un campo de cabecera que se puede emplear para solicitar al servidor un cambio de protocolo a nivel de aplicación (nunca a nivel de transporte) en la conexión que se está empleando. Por tanto, no se asegura que el servidor acepte el cambio, ni el cambio se hará en otra de las conexiones que pueda tener en paralelo el cliente con el servidor, ni mucho menos será posible cambiar, por ejemplo, de TCP a UDP. Sí será

posible, y de hecho son las aplicaciones que tiene esta cabecera, securizar conexiónes HTTP pasándolas a HTTPS (HTTP sobre TLS) o de HTTP a WebSocket (que se verá en profundidad en el último tema). La existencia de esta cabecera obliga a que la cabecera `Connection` también esté presente y que su valor sea `Upgrade`.

- en ocasiones se puede pedir al servidor, a través de la cabecera `Expect` que confirme que se va a comportar de la manera que se espera a determinadas peticiones del cliente.

El final de la cabecera lo marcan los caracteres de control <*CRLF*> que deben incluirse siempre. A partir de ahí comienza, si existiese y el método empleado lo permite, del cuerpo del mensaje. En él se transmite al servidor la entidad asociada al método de la petición. Por ejemplo, en un POST pueden ser los parámetros que necesite la aplicación web. Se sabe de la existencia del cuerpo del mensaje a través de la inclusión de una o dos cabeceras que son obligatorias cuando está presente: `Content-Length` y/o `Transfer-Encoding`.

`Transfer-Encoding` es una propiedad del mensaje, no de la entidad que se envía en sí y se emplea para indicar la transformación o transformaciones a las que se ha sometido a dicha entidad para su correcta transmisión. Distintos valores de esta cabecera señalan cómo se envía el mensaje: tal cual (con `identity`), comprimido (con `gzip`, `compress` o `deflate`) o dividido en trozos (con `chunked`). Para los dos primeros casos, o en caso de que no haya, es necesario dar a conocer al servidor el tamaño del cuerpo del mensaje, no de la entidad a enviar. Por tanto, este valor se calcula una vez aplicada la transformación elegida. Sin embargo, en el caso de la transmisión *chunked encoding*, el mensaje se divide en fragmentos de octetos que comienzan con una línea que contiene el tamaño en hexadecimal del mismo. Para indicar el fin de los bloques, se añade un *chunk* de tamaño 0.

Opcionalmente, se puede incluir la cabecera `Trailer`. En ella se listan las cabeceras que se podrán encontrar en el denominado *Trailer* que es la parte que se transmite después del mensaje, al recibir todos los *chunks* y antes de que se indique fin de la petición con un <*CRLF*>. Normalmente esto se emplea para enviar una suma MD5 del mensaje, en la cabecera `Content-MD5` del *Trailer*, que se ha enviado en el cuerpo de la petición. Cabe destacar que las cabeceras `Transfer-Encoding`, `Content-Length` y la propia `Trailer` no pueden estar contenidas en el *Trailer*.

Sirva de ejemplo de petición la siguiente:

```
POST /login.php? HTTP/1.1
Accept-language: en-US,en;q=0.8,es;q=0.6
Host: nodejskoans.com
Referer: nodejskoans.com
Content-type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml;q=0.9,*/*;q=0.8
Content-Length: 36

username=arturo&password=nodejskoans
```

5.1.2. La parte del Servidor

En la parte del servidor, las respuestas que éste envía al cliente como resultado de una petición, como sucede con las peticiones, también son líneas de texto claro que tienen una estructura determinada. La primera línea, *status-line*, contiene el código de respuesta, *status-code*, en una estructura como la que sigue:

<VERSION> <STATUS-CODE> <REASON PHRASE>

El campo relativo a la <VERSION> del protocolo se rige de la misma manera que en la petición

<STATUS-CODE> es un número de tres dígitos que define qué tipo de respuesta se está proporcionando. Hay cinco categorías, dependiendo del resultado de la petición en el servidor, enumeradas por el dígito de mayor peso, que indican [49, Section 6.1.1]:

1xx: información La petición ha sido recibida y entendida, y se procede a procesarla informando al cliente sólo con la *status-line* y alguna cabecera opcional. En HTTP 1.1 sólo hay dos códigos:

`100 Continue`
: sirve para informar al cliente de que el servidor acepta el tipo de petición en base a las cabeceras que le está enviando y puede continuar transmitiendo el cuerpo del mensaje. Es útil, por ejemplo, en el caso de que el cliente tenga que asegurarse de que el servidor va a aceptar datos que suponen un uso importante de ancho de banda o consumen

mucho tiempo en su transmisión. En este caso, el cliente empleará una cabecera `Expect: 100-continue`

`101 Switching Protocol` es la respuesta a una petición con la cabecera `Upgrade`. Expresa la conformidad del servidor para realizar el cambio de protocolo de aplicación a los que se solicitan, de manera inmediata al envío de esta respuesta. El servidor también puede forzar al cliente a solicitar el cambio de protocolo a través del código de respuesta `426 Upgrade Required`.

2xx: éxito

La petición fue recibida y procesada. Todo ha ido como se esperaba.

El código más típico de esta categoría es `200 OK`, que se obtiene cuando todo ha sido satisfactorio y se devuelve el recurso resultante del método. Si es éste es GET, se devuelve el recurso y, si es POST, el resultado de la acción.

3xx: redirección

La petición no se ha completado y se deben realizar más acciones para hacerlo. Generalmente indican un cambio de localización del recurso. La nueva localización se indica en una cabecera para ese propósito: `Location`.

Si el cambio es permanente y se ha asignado al recurso una nueva URI, se emplea `301 Moved Permanently` y para peticiones siguientes deben usarse la ubicación o una de las ubicaciones proporcionadas en `Location`.

Si por el contrario el cambio es temporal existen los códigos `302 Found`, `303 See Other` (sólo HTTP/1.1) y `307 Temporary Redirect` (sólo HTTP/1.1), con los que las siguientes peticiones deben seguir haciéndose a la URI original de la petición o a la que indica `Location`.

Normalmente estos códigos son transparentes al usuario ya que es el Agente de Usuario el que los interpreta y realiza la acción procedente.

4xx: error de cliente

La petición que se ha realizado está mal formada o contiene errores. Hay muchos casos por los que la petición puede ser errónea, siendo el más común `404 Not Found` con el que se indica que el servidor no ha encontrado el recurso donde indica la URI de la petición y no se sabe si se ha movido, ya que en ese caso se enviaría un *3xx*.

Sin embargo, el motivo más genérico de error es el `400 Bad Request` que indica que la sintaxis de la petición HTTP es errónea. El Agente de Usuario debería revisar que se hayan formado bien la *request-line* y las cabeceras, y que no contengan errores en sus nombres o valores.

Otros motivos de error pueden estar relacionados con los permisos de acceso al recurso. Si el recurso está protegido y el usuario no presenta credenciales o éstas son erróneas, se obtendrá un error `401 Unauthorized` y se pedirán las credenciales incluyendo obligatoriamente en la respuesta la cabecera `WWW-Authenticate`. Un Agente de Usuario se acredita mediante el campo `Authorization` en las cabeceras de la petición. El proceso de Autenticación y Autorización está descrito con detalle en la *RFC2617: HTTP Authentication: Basic and Digest Access Authentication*.

A grandes rasgos, cuando es necesario verificar la identidad del cliente que accede a determinados recursos, el servidor propone al Agente de Usuario que acredite que está autorizado presentando su contraseña a través de un esquema de autenticación. Éste se indica en la cabecera y puede ser `Basic` o `Digest`. Además en la misma cabecera se incluye el campo `realm` que es un texto que se presenta al usuario y le ofrece una pista sobre qué contraseña emplear.

`Basic` es un esquema en desuso porque el Agente de Usuario envía el par nombre de usuario y contraseña en la cabecera `Authorization` de la petición en claro, sin cifrar, sólo codificado en `base64`, y no es recomendable salvo, quizás, en conexiones seguras HTTPS.

El esquema `Digest` tampoco ofrece una seguridad alta, pero transmite la contraseña cifrada con un algoritmo criptográfico de dispersión que por defecto es `MD5`. En su modo más elemental, la cabecera incluye, aparte del modo, el campo `realm`, como en el esquema `Basic`. A partir de la recepción de esta respuesta, el cliente incluirá en las peticiones la cabecera `Authentication`, que consta de varios campos separados por comas y formados por pares clave/valor. Uno de estos campos es `response` y contiene la computación criptográfica de la *password*.

En ocasiones el recurso estará protegido y además el servidor no aceptará credenciales, aunque sean válidas. En ese caso el error devuelto es `403 Forbidden`, aunque si se quiere ocultar el recurso completamente, se puede

obtener un `404 Not Found`.

5xx: error de servidor
: La petición es correcta pero el servidor ha fallado al procesarla. Los distintos motivos se indican con un *status-code* concreto, por ejemplo, `500 Internal Server Error` avisa de un error indeterminado o `501 Not Implemented` indica que el servidor no soporta el método de la petición, entre otros.

La <REASON-PHRASE> es una breve explicación textual, comprensible por un humano, del código de respuesta. La RFC da recomendaciones sobre ellas, pero son opcionales y pueden cambiarse como se estime oportuno.

Las siguientes líneas a la *status-line* son las cabeceras de la respuesta, y su propósito es análogo al de las cabeceras de la petición: proveen información extendida de la respuesta.

Una cabecera obligatoria, que el servidor debe incluir siempre (con la excepción de los códigos *1xx* principalmente), es `Date` y estampa la fecha de la creación de la respuesta por parte del servidor. Su principal utilidad es la gestión por parte de la *chaché* de las entidades que se transmiten en el mensaje.

La fecha se puede expresar en varios formatos pero el preferido es el que se define en la *RFC1123: Requirements for Internet Hosts – Application and Support*, que incluye el día de la semana, día, mes, año y hora (hh:mm:ss) en la zona correspondiente. Por ejemplo: *Sun, 06 Nov 1994 08:49:37 GMT*.

Otra de las cabeceras es `Set-Cookie`. Con ella el servidor inicia el mecanismo que tiene HTTP para almacenar en la parte cliente pequeñas piezas de información referentes al estado de la comunicación HTTP. Estas piezas de información se conocen con el nombre de *cookies* y contienen pares `<clave>=<valor>;`. Cada *cookie* se fija con una cabecera `Set-Cookie`, que además de la propia *cookie* incluye la fecha de expiración de la misma, por lo que puede haber varias de éstas en una respuesta. Cuando un Agente de Usuario recibe esas cabeceras, almacena las *cookies* y está obligado a incluirlas (si no han expirado) en las siguientes peticiones al dominio, mediante la cabecera de petición `Cookie`.

Como parte del mecanismo de `Upgrade`, el servidor también puede incluir la cabecera homónima `Upgrade` en la respuesta de información de cambio de protocolo (`101 Switching Protocol`). Esta cabecera contendrá la pila de protocolos que se va a empezar a usar separándolos por comas. El primero de ellos se corresponde

con el que se usará en el nivel más bajo de la pila.

Un ejemplo más de cabecera de respuesta es la ya comentada `WWW-Authenticate`.

La última parte de una respuesta es la denominada *Entidad* (*Entity*) que es el contenido que se quiere transmitir como resultado de la petición, normalmente el recurso en sí. La *Entidad* añade unas cabeceras (*cabeceras de entidad*) a la respuesta con información acerca del contenido que se envía. Éstas principalmente hablan de las características del contenido, entre ellas:

Content-Encoding
: indica, cuando aparece, la codificación adicional a la que se ha sometido el contenido y por tanto, cómo debe el Agente de Usuario decodificarlo. Habitualmente se trata de compresión con distintos algoritmos (*zip*, *lempel-ziv-welch*...) que se indican con los valores `gzip`, `compress`, `deflate`...

Content-Length
: el tamaño en octetos del recurso que se envía. Si se le han aplicado codificaciones adicionales (indicadas con la cabecera anterior), el tamaño debe calcularse después de aplicarlas.

Content-Type
: expresa el formato MIME de la entidad que se envía de la misma manera en que MIME definió la cabecera del mismo nombre en la RFC1341.

Como ejemplo, se puede observar la respuesta HTTP a la petición de ejemplo anterior:

```
HTTP/1.1 200 OK
Content-Encoding: deflate
Content-Type: application/octect-stream
Date: Mon, 04 Mar 2013 17:23:34 GMT
```

5.2. HTTP en Node

Se tienen disponibles las clases que modelan un escenario HTTP básico importando el módulo http de la manera habitual:

```
var http = require('http');
```

El módulo comprende ambos extremos de una comunicación HTTP además de los mensajes que se intercambian las partes. El extremo servidor ofrece la clase http.Server

Se puede levantar un servidor HTTP invocando al método `http.createServer()` o instanciando directamente la clase *http.Server* (que es lo único que hace `createServer()`)

```
var httpserver = http.createServer([function(request, response){}]);
```

El servidor HTTP que ofrece este módulo se puede considerar como un Servidor TCP con nuevas funcionalidades o características ya que realmente hereda de la clase *net.Server*[2]. Consecuentemente, se puede atender a los eventos de ésta y utilizar sus métodos, incluido `listen()`, para empezar a escuchar conexiones entrantes, y `close()` para dejar de escucharlas, que son los más relevantes. Todo ello está documentado detalladamente en el capítulo anterior.

Ahora no sólo se pueden limitar el número de conexiones simultaneas sino que se pueden limitar el número de cabeceras de las peticiones para cada una de las conexiones con `httpserver.maxHeadersCount`.

El resto de las novedades que introduce el Servidor HTTP son todo eventos, principalmente referidos al ciclo de vida de las peticiones:

- con `net.Server` se sabía cuándo se producía una nueva conexión escuchando el evento `'connection'`. Ahora, la clase `http.Server` emite un evento `'request'` por cada petición HTTP que se recibe por las conexiones. Hay que tener en cuenta que a través de cada una de las conexiones se pueden recibir múltiples peticiones y, por tanto, cada conexión emitirá múltiples eventos `'request'`.

  ```
  httpserver.on('request', function(request, response){
    // procesar la peticion con opcion de responderla
  });
  ```

  El *callback* del *listener* puede definirse de esta manera o pasándolo opcionalmente a `http.createServer()`. Recibe como argumentos la petición que genera el evento, `request`, y la respuesta, `response`, que se configurará como se considere necesario y se enviará de vuelta al cliente, junto con el

[2]http.js: https://github.com/joyent/node/blob/v0.8.20-release/lib/http.js#L1750

cuerpo del mensaje si procede. Ambos mensajes son instancias de las clases *ServerRequest* y *ServerResponse* respectivamente, que se tratan más adelante, y que básicamente son un modelo de las peticiones y la respuesta tal y como las definen las RFCs relacionadas con HTTP.

Para generar el objeto `request` lo que Node hace internamente es instalar un *listener* en el evento `'connection'`[3] para capturar el *socket* de la conexión y monitorizar todo lo que se recibe desde el cliente para procesarlo con un *Parser HTTP* de un *pool* de *Parsers*. El *Parser HTTP* se obtiene del *binding* `node_parser` y es quien internamente crea el objeto `request`.

El objeto `response` se genera una vez se han procesado las cabeceras HTTP de la petición[4]. Ambos, `request` y `response`, se entregan a la aplicación mediante el evento `'request'` que se encarga de emitir el Servidor.

- los errores en cualquier conexión que haya en el servidor se notifican a través de `'clientError'` que aportará al *callback* información de la excepción generada:

```
httpserver.on('clientError', function(excepcion){ });
```

Además, para cada petición, el servidor detecta automáticamente los mecanismos de *Continue* y *Upgrade* que se notifican con los eventos `'checkContinue'` y `'upgrade'` respectivamente. En ambos casos, si no se atiende el evento, la conexión se cierra automáticamente.

`'checkContinue'` se emite cuando un cliente solicita confirmación para continuar con la petición a través de la cabecera `Expect: 100-continue`, inhibiendo la emisión del evento `'request'`. Si éste se atiende con `server.on('checkContinue', function(request, response){ })` debe generarse adecuadamente la respuesta, empleando métodos específicos de `response` si se admite el cuerpo de la petición, o enviando el código de respuesta que se considere oportuno, por ejemplo, un `417 Expectation Failed`, si se rechaza.

`'upgrade'` se emite cuando un cliente solicita una actualización de protocolo. La acción a realizar en este caso se define como un manejador del evento, `server.on('upgrade', function(request, socket, head){ })`.

[3]http.js: https://github.com/joyent/node/blob/v0.8.20-release/lib/http.js#L1748
[4]http.js: https://github.com/joyent/node/blob/v0.8.20-release/lib/http.js#L1868

Como cuando el cliente pide *Upgrade* es para cambiar el protocolo de aplicación que se está empleando en la conexión, Node pone a disposición del *listener* el *socket* entre cliente y servidor, con el que se puede trabajar directamente, y un *Buffer*, `head`, que puede contener el primer paquete que el cliente envía con el nuevo protocolo. Este paquete encapsula cabeceras y datos y es deber del programador interpretarlo correctamente para identificar su contenido. El resto de paquetes se reciben a través del *socket* de la manera habitual en que se reciben datos en un *Stream*: atendiendo al evento `'data'`:

```
socket.on('data', function(data){ });
```

Otro de las características de HTTP 1.1 que Node soporta específicamente es el método CONNECT, a través del evento del mismo nombre, `'connect'`. Éste se emite cuando se recibe una petición con dicho método y, de manera análoga al evento anterior, se puede escuchar con `server.on('connect', function(request, socket, head){ })` donde los argumentos que se reciben y su función son idénticos al del caso de `'upgrade'` y, también de la misma manera que antes, se pueden recibir los paquetes que el cliente envía por el túnel hacia el otro extremo: instalando un manejador del evento `'data'` en el *socket*.

Se debe observar que si se desea envíar una respuesta HTTP, ésta no se recibe como argumento en los *listeners* sino que debe ser escrita directamente en el `socket`.

5.2.1. ServerRequest

El objeto *ServerRequest* modela la petición HTTP que el cliente realiza al servidor y al que se accede mediante los eventos exclusivos de `http.Server`. *ServerRequest* implementa un *Stream* de tipo *readable* lo cual tiene toda lógica puesto que la petición viene del cliente y no tiene sentido que se pueda modificar, para ello está la respuesta.

El API de Node advierte que es una clase que no se puede instanciar directamente. Y en efecto, no se puede instanciar con `new` ya que, en detalle, los objetos de esta clase los crea `node_http_parser` bajo demanda[5].

ServerRequest proporciona métodos para obtener toda la información posible de la petición, empezando por el propio *socket* por donde se envía

[5]http.js: https://github.com/joyent/node/blob/v0.8.20-release/lib/http.js#L1793

`request.connection`, y siguiendo por la imprescindible *request-line*:

`serverRequest.method` es una *String* inmutable que contiene el nombre del método

`serverRequest.url` contiene la URI de la petición, que se recomienda parsear adecuadamente con el método del módulo *url* `require('url').parse()`.

`serverRequest.httpVersion` da la versión del protocolo en una *String* que contiene exlusivamente el número

Las cabeceras de la petición están contenidas todas en el objeto JSON `serverRequest.headers` y son accesibles teniendo en cuenta que las propiedades de este objeto son el nombre de las cabeceras en minúscula.

Si la petición contiene cuerpo de mensaje, éste será accesible gracias a que la petición es un *Stream* y los datos que se envían a través del *socket* pueden recibirse atendiendo al evento `'data'` de la manera habitual:

```
serverRequest.on('data', function(datos){  });
```

Esto implica que se tienen a disposición el resto de métodos y eventos de la clase *Stream*: `pause()` o `resume()` y `'end'` o `'close'`.

Se debe contemplar el caso que la transferencia se haga por *chunks*, y se sigan las reglas de una `Transfer-Encoding: chunked`. Si es así, es posible que existan cabeceras adicionales contenidas en el *Trailer* de la petición, a las que se podrá acceder gracias a la propiedad `request.trailers`, que es un objeto JSON conteniéndolas todas. No obstante, por su naturaleza, el *Trailer* sólo estará disponible una vez se haya recibido toda la petición y, por tanto, se haya emitido el evento `'end'`.

5.2.2. ServerResponse

El objeto *ServerResponse* modela la respuesta HTTP que el servidor envía al cliente. Va íntimamente relacionada con la petición que la causa, relación que se refleja en el código del módulo porque para instanciar un objeto *ServerResponse* es necesario que esté ligarlo a un objeto petición `request` del que se ha procesado las cabeceras[6]. De todas maneras, una *ServerResponse* se generará automáticamen-

[6]http.js: https://github.com/joyent/node/blob/v0.8.20-release/lib/http.js#L1868

te y será puesta a disposición del programador a través de uno de los múltiples eventos de un servidor `http.Server`. *ServerResponse* implementa un *Stream* de sólo escritura, como es lógico para configurar una respuesta adecuada que enviar al cliente.

Con este objetivo se ofrecen facilidades, para generar desde las cabeceras hasta el cuerpo, con soporte para el mecanismo de *Continue* gracias al método `response.writeContinue()`, que símplemente envía el mensaje `HTTP/1.1 100 Continue` al cliente.

Construir una respuesta HTTP se puede hacer ordenadamente siguiendo unos pasos:

- Generar la *response-header*. El método que puede usarse es `serverResponse.writeHead(statusCode, [reasonPhrase], [cabeceras])`. Sólo se puede invocar una única vez y antes de cualquier otro método de respuesta. A través de él se fija el código de respuesta con el argumento `statusCode`. Opcionalmente pueden fijarse las `cabeceras` en notación JSON.

  En realidad, no es estrictamente necesario llamar a este método para construir una respuesta. Se puede directamente escribir con `serverResponse.write()` o `serverResponse.end()`. Las cabeceras necesarias, denominadas por el API "cabeceras implícitas" (porque se han generado sin llamar a `serverResponse.writeHead()`), se calculan y se añaden automáticamente. Si hubiera necesidad de realizar operaciones con estas cabeceras implícitas existen una serie de métodos para ello, que se invocarán antes de llamar a `serverResponse.write()`:

  - los accesores `server.Response.getHeader(cabecera)` y `responseHeader.setHeader(cabecera,valor)` para obtener y para asignar una `cabecera` respectivamente
  - `serverResponse.removeHeader(nombre)` para eliminar una cabecera

  En los casos en que la respuesta se genera implícitamente, se puede asignar el *status code* fijando la propiedad numérica `serverResponse.statusCode` antes de enviarse las cabeceras. Si ya se han enviado, contendrá el *status code* que se envió.

- completar el contenido del cuerpo de la respuesta con

`serverResponse.write()`

- en ocasiones, en una respuesta enviada en bloques o *chunks* puede desearse añadir un *Trailer* con las cabeceras adicionales permitidas. En este caso, se puede hacer uso de `serverResponse.addTrailers()` cuyo único argumento es un objeto JSON de propiedades las cabeceras que se deseen incluir. Si se invoca este método en una transferencia que no sea `chunked`, no tendrá ningún efecto.
- dar por finalizada la respuesta y dejarla lista para el envío con `serverResponse.end()`

5.2.3. Clientes HTTP

Con Node se pueden realizar peticiones HTTP a un servidor a través del método `http.request(opciones, function(){})`

El argumento opciones puede adoptar dos formas. La más sencilla es la de un String que es la URL del recurso al que se pretende acceder.

El segundo formato que se le puede dar a opciones es el de un objeto JSON a partir del cual se configura la conexión. Como siempre, ésta quedará definida principalmente por la IP o el nombre de dominio del servidor al que se conecta (`opciones.hostname` u `opciones.host`), su puerto (`opciones.port`) y el interfaz local desde el que se realiza (`opciones.localAddress`). Sobre dicha conexión se realizará una petición, instancia de la clase *ClientRequest* que a su vez es un *Stream* de sólo escritura. Este objeto representa la petición en curso y se controlará la evolución de la comunicación HTTP manipulándolo a través de sus métodos.

Cuando la conexión ha sido creada, *ClientRequest* notifica al programador, mediante el evento `'socket'`, que dispone de un *socket* sobre el que realizar el intercambio petición/respuesta y que está listo para ello:

```
clientRequest.on('socket', function(socket){ });
```

No es necesario atender a este evento para continuar la petición, sino que simplemente es una notificación más de las fases por las que va pasando la comunicación.

Una vez preparado, no sería necesario escribir la cabecera completa en el *socket*, ya que la *request-line* y algunas cabeceras de la petición se crean automáticamente cuando se invoca a *http.request()*, empleando valores de las opciones que se le pasan. En este caso, la *request-line* se configura: fijando el método de la petición se a través de `opciones.method`, por defecto `'GET'`, obteniendo la url del recurso, incluidas *queries* si procediese, de `opciones.path`, por defecto '/' empleando siempre la versión del protocolo 1.1[7].

El resto de cabeceras también se fijan opcionalmente con el objeto JSON `opciones.headers`, aunque algunas se obtienen de otras opciones, como `Host`, que sale de `opciones.hostname` o de `opciones.host` (si existen las dos, `hostname` es la preferida por motivos de compatibilidad con el módulo *url*), o `Authorization`, que se computa con `opciones.auth`.

La cabecera completa se queda en cola de salida para ser enviada cuando comience a generarse el cuerpo de la petición, si lo hubiera. Mientras ésta no se haya enviado, las cabeceras pueden modificarse con los métodos accesores `getHeader(nombre)`, `setHeader(nombre, valor)` y `removeHeader(nombre)`.

En caso de que haya una entidad que precise enviarse, se hará con el método que el interfaz *Stream* define: `clientRequest.write(datos, [codificacion])`.

Completada la petición, tanto si se ha escrito en el *stream* como si no, se debe indicar con `clientRequest.end()` a menos que se desee cancelar todo el proceso con `clientRequest.abort()`.

El siguiente paso en la interacción es esperar a que el servidor envíe su respuesta y procesarla. Siguiendo la filosofía de la plataforma, la manera de recibirla es a través de eventos que `clientRequest` puede atender instalando para ellos los correspondientes *callbacks*. Típicamente la respuesta será de aquellas con los códigos de respuesta y las cabeceras más habituales, que indican que se ha atendido satisfactoriamente o que ha habido algún error, y con alguna entidad como cuerpo de mensaje. Todo esto se recibe por el cliente como un objeto, instancia de *ClientResponse* (tratado más adelante), a través del evento `'response'`, que solamente se dispara una vez:

```
clientRequest.on('response', function(response){ });
```

El resto de eventos tratan los casos concretos de los mecanismos ya conocidos de

[7]http.js: https://github.com/joyent/node/blob/v0.8.20-release/lib/http.js#L1294

`Continue`, `Upgrade` o `Connect`, como se ha visto que suele hacerse a lo largo de toda la librería.

Para las respuestas informativas, del grupo *1xx*, se dispararán los eventos `'continue'` y `'upgrade'`. En el primer caso como resultado de la recepción de un código de respuesta `100 Continue`, y en el segundo, de un `101 Switching Protocols`. En ambos, lo normal es que estos códigos sean el resultado de haberlos solicitado al servidor, como parte de la negociación, con las cabeceras `Expect: 100-continue` o `Upgrade`. Sin embargo no es condición indispensable: si, sin incluirlas, el servidor por cualquier motivo sí los retornase, el evento se activaría igualmente. No obstante, esta situación es anómala y no debería producirse en el transcurso de la comunicación.

La acción a seguir la determina como siempre la función que atiende el evento. En caso de un `100 Continue`, no se necesita más información adicional para proseguir con la ejecución del programa y, por tanto, el listener no recibe ningún argumento:

```
clientRequest.on('continue', function(){ });
```

Por el contrario, en respuestas `101 Switching Protocol`, es necesario tener una referencia al *socket* de la conexión, con objeto de enviar el nuevo protocolo por él, y también tener disponibles los primeros datos que vienen sobre él, si el servidor los hubiera empezado a enviar:

```
clientRequest.on('upgrade', function(response, socket, datos){ });
```

El caso del evento `'connect'`, disparado durante el transcurso del mecanismo *Connect*, es distinto: para que este evento se emita y pueda ser atendido es necesario que se haya realizado una petición con el método CONNECT que puede ser contestada con un *status-code* cualquiera por parte del servidor. Este código lo recibirá el *listener* del evento, junto con el `socket` por el que se desarrolla la comunicación entre ambos extremos de la conexión y los primeros datos `connectHead` que se reciben (si es que el extremo opuesto enviara):

```
clientRequest.on('connect', function(res, socket, connectHead){};
```

A diferencia de casos anteriores, el tipo de respuesta debe ser procesado por el programa y el desarrollador debe contemplar los casos que le interesen. Como ejemplo, si el servidor devolviese un `101 Switching Protocol`, esta vez no se emitiría ningún evento `'upgrade'`, sería el programa el que tendría que decidir

cómo manejarlo. Aún así, este caso es extraño y tampoco debería producirse en una comunicación normal.

5.2.4. ClientResponse

ClientResponse modela la respuesta del servidor que recibe el cliente como un *Stream* de sólo lectura con métodos y eventos para manejar su ciclo de vida. Se puede ver como el equivalente en la parte de cliente del objeto *ServerRequest*, aunque se corresponde con la *ServerResponse* que genera el servidor.

Es un objeto que *ClientRequest* recibe como resultado de atender algunos de los eventos definidos para él como `'response'`, `'upgrade'`, `'continue'` o `'connect'`. Por tanto, va asociado al mismo *ClientRequest*, no pudiendo ser instanciado directamente: su constructor necesita como argumento la petición `req` a la que está ligada.

Permite obtener la información necesaria de cada una de las partes de la respuesta. Para empezar, se accede a los campos de *status-line* con un par de propiedades inmutables:

- la versión se consulta a través de `clientResponse.httpVersion`, que es idéntica a su homónima en `serverRequest.httpVersion`
- `clientResponse.statusCode` contiene el número de tres dígitos del código de respuesta

El resto de cabeceras se encuentran en la propiedad `clientResponse.headers`, siguiendo la línea marcada por la librería. Al igual que las cabeceras contenidas en el *Trailer*, accesibles desde `response.trailers` una vez recibida toda la respuesta del servidor.

Si la respuesta incluye una entidad, ésta estará disponible conforme se vaya recibiendo, de la manera en que se leen los datos en un *Stream*: con el evento `'data'`.

```
clientResponse.on('data', function(datos){ });
```

Por supuesto, los eventos (`'end'`, `'close'`) y métodos (`pause()`, `resume()`, `setEncoding()`) propios de un *Stream* de sólo lectura también están disponibles para su utilización.

5.3. Aplicación con HTTP

En esta práctica se dotará al servidor RTP de una interfaz web que controle la lista de reproducción de una IP de *broadcast* concreta, protegida por contraseña. Se presentará en una página unos sencillos botones de reproducción, pausa y adelante/atrás junto con una lista de las canciones disponibles. Los efectos de estas acciones sobre la lista de canciones se notarán en todos los clientes que pertenezcan al grupo de *broadcast*, con lo que puede usarse como panel de administración de la difusión del contenido.

5.3.1. Descripción del problema

Los requisitos básicos de la aplicación se describen a grandes rasgos a continuación. Se debe tener en cuenta que el funcionamiento es realmente sencillo.

El cliente administrador deberá ser capaz de obtener la interfaz a través de un navegador web moderno (por tanto, estará programada en *html*), y ésta debe presentar un formulario cuyos botones realicen las acciones requeridas: play (o pause en caso de que ya se esté reproduciendo), siguiente pista y pista anterior. El resultado de pulsar uno de dichos botones devolverá la interfaz web actualizada con el estado actual del reproductor.

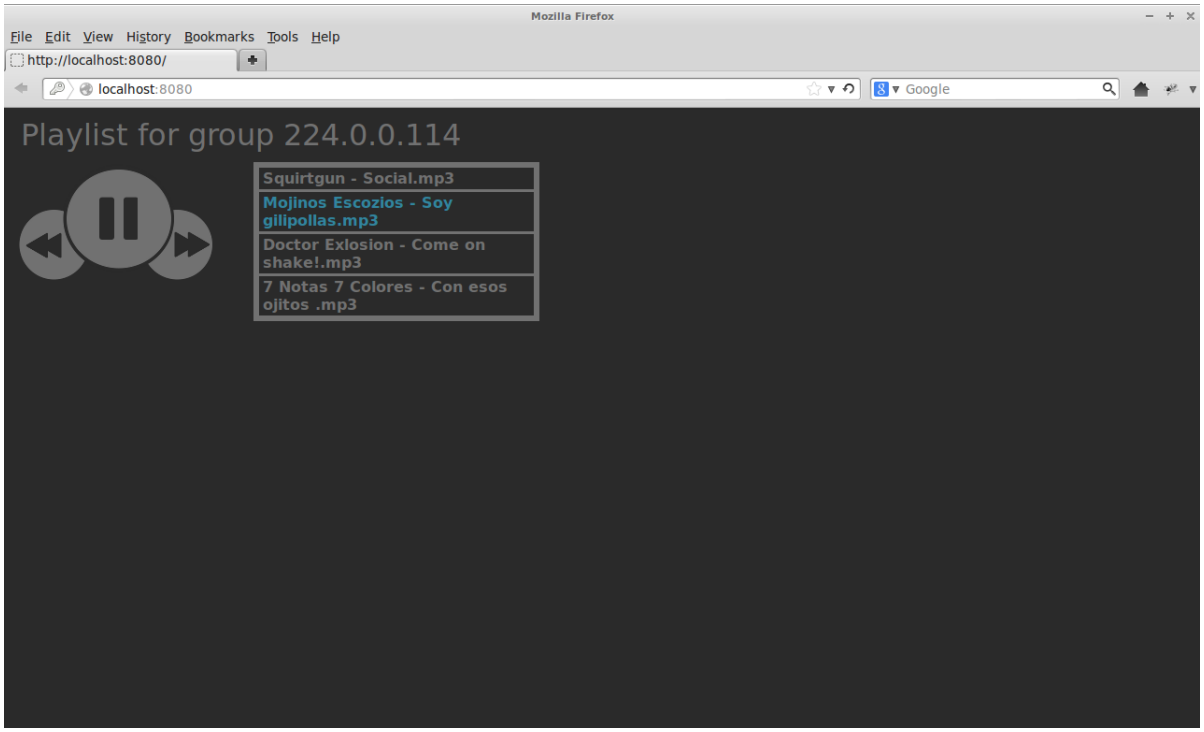

Figura 5.1: Interfaz de la Solución HTTP

Para que sólo el administrador de la lista de canciones en una determinada IP tenga acceso a la interfaz web, ésta estará protegida por contraseña. El nombre de usuario al que se asocia la contraseña, por convención, será la IP de *broadcast* sobre la que se actúa. El servidor de listas de reproducción debe determinar si esta contraseña para la IP a la que se está transmitiendo la lista en *streaming* es correcta y, si procede, enviarle la interfaz.

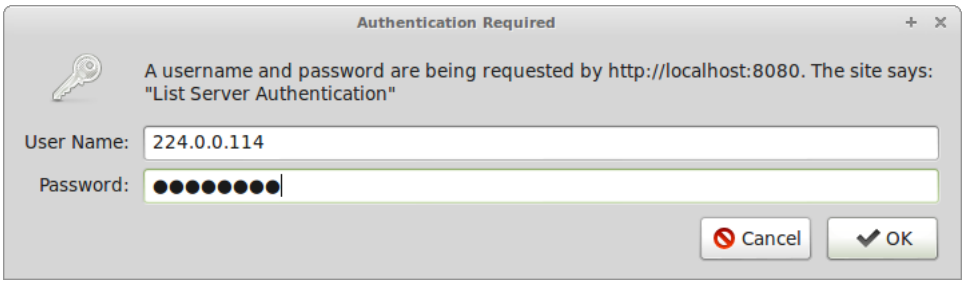

Figura 5.2: Autenticación en la Solución HTTP

5.3.2. Diseño propuesto

Esta práctica es parecida a la práctica anterior, por lo que habrá similitudes en la forma que tienen los distintos módulos de la aplicación para comunicarse entre ellos. Con la intención de separar las distintas partes de la aplicación, se propone aislar la funcionalidad del servidor HTTP de Listas de Reproducción en el módulo `ListHttpServer.js` que se relaciona con el resto de módulos del sistema según el siguiente esquema de la arquitectura:

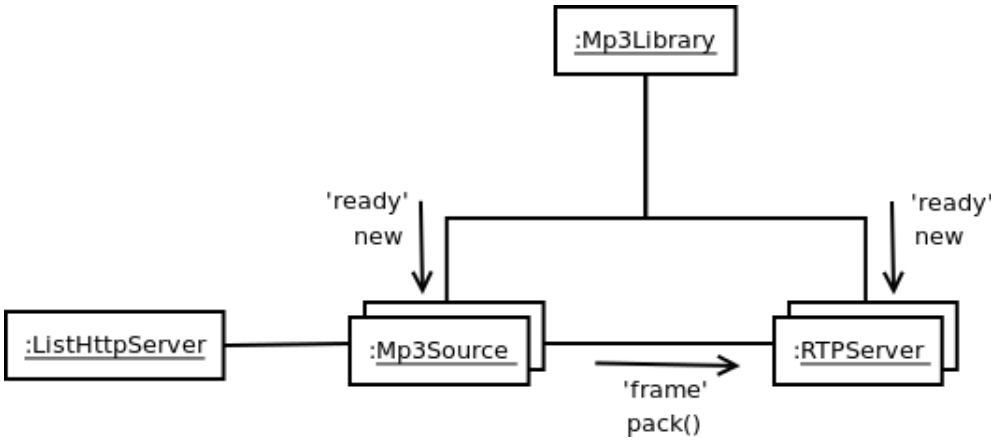

Figura 5.3: Diagrama de colaboración de la Solución HTTP

El módulo que interesa, `ListHttpServer.js`, proporcionará una función específica para la creación de este servidor, `create()`, que simplemente devuelve un servidor HTTP preparado para escuchar los eventos concretos necesarios para la práctica y capaz de atender lo que solicitan devolviendo la respuesta correcta al cliente.

Como la finalidad de un objeto *ListHttpServer* es controlar el Reproductor asociado a una IP *broadcast*, será necesario proporcionarle una lista de IPs de *broadcast* con los reproductores asociado a ellas. De esta manera con un sólo servidor *ListHttpServer* se gestionan los reproductores de todas las listas:

```
var listServer = http.createServer();
exports.create = function(db){
  if (db == null) throw new Error('Database cannot be empty');
  listServer.broadcastList = db;
  return listServer;
}
```

La interfaz del Reproductor es sencilla, posee un método por cada acción que se quiera realizar sobre él (play, pause...). Los métodos admiten una función de *callback* que se ejecutará cuando se haya terminado de realizar la acción, de esta forma, por ejemplo, se podrá enviar la respuesta al cliente con la seguridad de que la acción que ha solicitado se ha llevado a cabo.

Como *ListHttpServer* sólo tiene como propósito procesar peticiones web con los métodos habituales, GET y POST, se atenderá exclusivamente al evento `'request'`:

```
listServer.on('request', function(req, res){
  switch(req.method){
    case 'GET':
      // GET requests processing code
      break;
    case 'POST':
      // POST requests processing code
      break;
  }
})
```

El código asociado al evento será el que procese el tipo de petición. Los requisitos que se exigen a esta aplicación implican solamente a dos de ellos:

- obtener la interfaz y elementos estáticos involucra peticiones de tipo GET.
- interactuar con la aplicación mediante un formulario, en este caso los botones del Reproductor, involucra peticiones de tipo POST.

Ante cualquier petición, la primera decisión a tomar es permitir o denegar el acceso a la página principal del reproductor, protegiéndola con una contraseña. Solamente el administrador de la lista de reproducción de la dirección de *broadcast* concreta debería ser capaz de controlar la lista. Se tomará por convención que el nombre de usuario sea la IP que se administra y la contraseña será `password`, a modo de ejemplo. Por ejemplo, si la emisión *broadcast* se realiza en la dirección `224.0.0.114`, habrá una tabla usuario/contraseña como la siguiente:

```
var allowedList = {
  "224.0.0.114": "password"
}
```

El tipo de autenticación a utilizar es la más sencilla: `Basic`. Cada vez que se reciba una petición, se comprobará pues que existe la cabecera `Authorization` y que las credenciales que se presentan en ella son las correctas. En caso de no serlo se devolvería el código de estado adecuado para este caso: `401 Unauthorized`.

```
var credentials = req.headers["authorization"];

var userAndPass, broadcastIp, pass;
if (credentials){
  userAndPass = new Buffer(credentials.split(' ')[1],
                           'base64').toString('ascii').split(':');
  broadcastIp = userAndPass[0];
  pass = userAndPass[1];
}

if (!credentials
    || !(broadcastIp in allowedList
         && broadcastIp in this.broadcastList)
    || allowedList[broadcastIp] != pass) {
      res.writeHead(401, "Unauthorized", {
        "WWW-Authenticate": 'Basic realm="List Server Authentication"'
      });
      res.end();
      return;
}
```

Si se determina que el usuario es el administrador legítimo de la lista, se procede a resolver qué solicita. Las opciones son muy limitadas teniendo en cuenta la sencillez de la aplicación. En este sentido, pueden pedirsese:

- la página web principal escrita en *html* que contiene la interfaz de control del Reproductor. Por ser el punto de entrada a la aplicación, la *request-line* esperada es `GET / HTTP/1.1`. Para identificar este mensaje, debe parsearse correctamente la *request-line* anterior y determinar que el path que se pide es `'/'`:

  ```
  var uri = url.parse(req.url);
  var path = uri.pathname;
  ```

Esta página debe ser generada dinámicamente utilizando el estado del reproductor (reproduciendo una pista concreta o pausado) en el momento de la petición. Se empleará un método privado del módulo, `writeDocument()`, creado para tal efecto.

`writeDocument(res, player)` escribe en el *Stream* de respuesta `res` los datos que se solicitan al reproductor `player`. El formato en que los escribe es *html* por lo que, si no ha habido ningún error durante el procesado, se debe fijar la cabecera `Content-Type` de la respuesta con el tipo MIME adecuado. `Content-Length` también debe establecerse:

```
var content = head + info + form + list + tail;
res.writeHead(200, 'OK', {
  "Content-type": "text/html",
  "Content-length": content.length
});
res.write(content);
res.end();
```

- algún fichero estático, como imágenes u hojas de estilo que apliquen un diseño sobre la página principal. Estas entidades se solicitan con una petición de tipo GET y se diferencian de las anteriores peticiones en que el *path* es distinto de '/'. La ruta en el sistema de ficheros es relativa al directorio donde reside el script ejecutable del servidor. Se debe tener muy en consideración que no se realiza ningún tipo de control sobre la ruta que se recibe en las peticiones por lo que la aplicación es totalmente vulnerable a ataques de *directorio transversal*.

  Si al buscar el fichero, se obtiene un error, se indicará con el código correspondiente, en este caso un `404 Not Found`, indicando que no se ha encontrado el recurso. Si, por el contrario, se ha localizado y se procede a enviarlo, se dejará que el código de respuesta lo genere Node implícitamente y lo único que se fijará serán las cabeceras `Content-Type` y `Content-Length` con `response.setHeader()`.

  ```
  fs.readFile("." + path, function(error, data){
    if (error){
      res.writeHead(404, 'Not found');
      res.end();
  ```

```
    return;
  }
  var fileExtension = path.substr(path.lastIndexOf(".") + 1);
  var mimeType = MIME_TYPES[fileExtension];
  res.setHeader("Content-Type", mimeType);
  if (mimeType.indexOf("text/") >= 0){
    res.setHeader("Content-Encoding", "utf-8");
  }
  res.setHeader("Content-Length", data.length);
  res.write(data, "binary");
  res.end();
});
```

- que se realice una acción sobre el reproductor. Proviene del formulario que presenta la interfaz *html*:

  ```
  <form method="post" action="/">
    <input type="submit" value="prev" name="action">
    <input type="submit" value="play" name="action">
    <input type="submit" value="next" name="action">
  </form>
  ```

  así que se espera una petición de tipo POST cuyo contenido es un único campo `action`. El valor de action, además de indicar qué acción tomar, coincide con el nombre del método a invocar del reproductor que se controla. Con esta convención, se pretende simplificar la manera de llamar al método, de tal forma que sólo sea cuestión de:

  ```
  if (action in player) {
    player[action](function(){
      writeDocument(res, {group:broadcastIp,
                          paused:player.paused,
                          tracks:player.list(),
                          currentTrack:player.currentTrack()});
    });
  }
  ```

Procesar esta petición implica que el servidor espera un cuerpo de mensaje de longitud conocida que, en condiciones normales, está disponible inmediatamente

en el servidor porque su tamaño es mínimo. Aún así, no debe darse la petición por recibida hasta que el cliente no la finalice. Por tanto se deben atender dos eventos:

- `'data'` para recibir los datos del formulario

```
var body = '';
req.on('data', function(data){
  body += data;
})
```

- `'end'` como indicador de que debe procesarse el formulario. Esto se traduce básicamente en interpretar la *query* recibida para lo que se hará uso del módulo *querystring* del core de Node.

```
req.on('end', function(){
  var action = querystring.parse(body).action;
  var player = self.broadcastList[broadcastIp];
  if (action in player) {
    player[action](function(){
      writeDocument(res, player)
    })
  }
})
```

Como resultado se espera la misma página web que se genera cuando se accede al reproductor, pero actualizada con el estado en que se encuentre el reproductor (parado, en el siguiente track...) después de la acción. A tal efecto será útil de nuevo la función `writeDocument()` que se invocará en el *callback* que se pasa como argumento al Reproductor, puesto que la página *html* se debe enviar una vez ejecutada la acción.

5.4. Objetivos de los Koans

A través de los Koans asociados al tema HTTP, se pretende obtener un conocimiento del módulo para la creación de sencillos servidores HTTP y cómo se abordan aspectos básicos del protocolo. Para la creación de aplicaciones más complejas existen módulos de terceras partes que hacen más fácil el manejo de

características más avanzadas de HTTP. Alguno de ellos se tratará en el tema siguiente. Por lo pronto, se analizarán los siete Koans del tema.

1. Saber que `'request'` es el principal evento que puede atender un servidor HTTP:

   ```
   server.on(___, function(req, res){})
   ```

2. Extraer las cabeceras de la petición que se necesiten del objeto `headers` propiedad del objeto `request`:

   ```
   var credentials = request.___['authorization'];
   ```

3. Conocer el módulo *url* y ser capaz de diseccionar las URLs que las peticiones contienen en la *request-line* con `parse()`

   ```
   var url = require('url');
   var uri = url.___(req.url);
   ```

4. Ser capaces de identificar qué método de la petición está empleando el cliente a través de la propiedad `method`

   ```
   switch(req.___){
     case 'GET':
       break;
     case 'POST':
       break;
   }
   ```

5. Generar una respuesta implícita incluyendo las cabeceras necesarias con `setHeader()`

   ```
   res.___("Content-Type", "text/css");
   ```

6. Conocer el módulo *querystring* y ser capaz de diseccionar con él la *query* de una petición

   ```
   var querystring = require('querystring');
   var query = querystring.___(body);
   ```

5.5. Preparación del entorno y ejecución de los Koans

Como es habitual, se requiere el módulo *koanizer*, proporcionado con la práctica, para poder ejecutar los Koans de manera satisfactoria.

El fichero que contiene los *koans* es `http-koans.js`, que es en realidad el módulo *ListHttpServer*, que contiene la totalidad de la lógica que trata con HTTP en la aplicación. Como en todas las prácticas anteriores debe ser editado y completado, sin que quede ningún hueco ___.

Para verificar que se han realizado con éxito los *koans* se ejecutan los casos de prueba con:

```
$ jasmine-node –verbose spec/
```

Como resultado de una correcta resolución de los mismos, la práctica será completamente funcional y se podrá ejecutar con el comando:

```
$ node app.js
```

Como en todas las prácticas que incluyen al protocolo RTP para generar contenido de audio en *streaming*, se debe configurar un Reproductor tal y como se indicó en el apartado "Descripción de la aplicación" del capítulo "Módulos dgram y buffer". Puesto que el objetivo de la práctica es la gestión del audio que se emite en un grupo concreto *multicast*, hay que volver a fijar la dirección IP a la del grupo que se ha tomado como ejemplo en la práctica: el `224.0.0.114` con puerto `5002`. Huelga volver a describir los pasos a seguir para hacerlo.

Para realizar operaciones sobre la lista, se debe apuntar un navegador a la URL `http://localhost:8080`, en caso de que se esté ejecutando en local. A partir de este punto el resto está suficientemente explicado a lo largo de los apartados anteriores.

5.6. Conclusión

En este capítulo se ha fijado el manejo a través del módulo *http* de las características de más bajo nivel del protocolo HTTP: recepción de peticiones atendiendo al evento `'request'` identificando en ellas los parámetros imprescindibles como el método empleado `method` o sus cabeceras `headers`. Con toda esta información se

ha podido ver cómo generar una respuesta adecuada, por ejemplo, estableciendo también unas cabeceras concretas con `setHeader()`.

Capítulo 6

Express

6.1. Connect

Connect es un *framework middleware*, lo que significa, en palabras de su propio autor [50], TJ Holowaychuk, que es *"una capa intermedia que proporciona a los desarrolladores de Node una solución efectiva de alto rendimiento para un rápido desarrollo usando componentes intercambiables denominados middleware"*. Estos componentes son módulos, semejantes al resto de los módulos de Node, que se pueden encadenar unos con otros en el orden que se desee formando una pila de módulos (*stack*). Este *stack* es la capa intermedia de la que se hablaba, porque se sitúa encima de Node y sus módulos y debajo de la aplicación.

El autor divide en dos categorías estos componentes:

Filtros (*filters*), que son módulos que se pueden colocar arbitrariamente en el *stack*, y que procesan el tráfico, tanto entrante como saliente, sin responder a las peticiones. El ejemplo típico es el *middleware log*, que registra el contenido de las peticiones sin responder, dejando que sea la aplicación la que provea esa lógica.

Proveedor (*provider*), representa un *endpoint* en el *stack*, lo que implica que la petición no progresa en caso de que deba ser atendida por el módulo. Por ejemplo, el *middleware json-rpc* sólo responde a peticiones cuyo `Content-Type` sea `application/json`, dejando pasar hasta nuestra aplicación el resto de peticiones.

Hasta la fecha, en su versión 2.7, Connect posee más de 20 *middlewares* que ofrecen una variada funcionalidad:

logger

Trazas de log, por defecto a la salida estándar pero redirigible a cualquier *Stream*. La información a registrar se ofrece a través de *tokens* a los cuales puede dárseles un formato de presentación. Tanto los *tokens* como el formato, aparte de haber algunos predefinidos, pueden ser personalizados por el desarrollador.

Por ejemplo, el formato `'tiny'` loguea el método, la URL, el *status*, el campo `Content-Length` de la respuesta y el tiempo en milisegundos que tarda, a través de sus correspondientes *tokens*:

:method :url :status :res[content-length] - :response-time ms

csrf

Prevención del ataque *Cross-Site Request Forguery* [51].

Mediante este ataque, una página maliciosa puede enviar una petición a un servidor vulnerable si conoce la URL y los parámetros de la *query* con los que realizar la petición. Normalmente la URL se inserta en la página maliciosa como un recurso, de tal manera que será el navegado el que la solicite sin intervención del usuario, por ejemplo:

```
<img src="http://bank.com/transfer?amount=1000&to=Malicious"/>
```

Si hay alguna sesión abierta con la página vulnerable, el navegador realizará la petición a través de esa sesión, ejecutándose con éxito el ataque.

Para mitigarlo en la manera de lo posible, una de las soluciones, que es la que implementa este *middleware*, consiste en generar un *token* por sesión de usuario, aleatorio e impredecible, que se almacenará en la sesión del usuario. Este *token* será uno de los campos de los formularios con los que se vaya a realizar una transacción. Si la transacción es legítima y se realiza dentro del marco de la sesión actual, el token del formulario y el almacenado serán idénticos. Si no, se generará un error `403 Forbidden` y se frustará el ataque.

Dependiente del *middleware*: *sesion* y *cookieParser*.

compress

Compresión `gzip` y `deflate` de la entidad que se transmite en la respuesta, a menos que se el campo `Content-Encoding` se haya definido como `identity` (sin compresión). Modificará o añadirá la cabecera `Content-Encoding`.

basicAuth

Soporte para la autenticación HTTP básica.

Admite por parámetros, o bien un usuario/password específico contra el cual realizar la validación o una función que realice el proceso de autenticación. Esta función es susceptible de contener código propio del desarrollador para la validación contra una base de datos por ejemplo, y devolverá `true` cuando el proceso haya sido sastisfactorio.

bodyParser

Procesa el cuerpo de las peticiones con soporte para los tipos MIME `application/json`, `application/x-www-form-urlencoded` y `multipart/form-data`.

En realidad es equivalente a utilizar los módulos *json*, *urlencoded* y *multipart* aunque no hay que incluirlos explícitamente en el *stack*.

Dependiente del *middleware*: *urlencoded*, *multipart* y *json*.

json

Procesa el cuerpo de una petición web en formato JSON, definido por su `Content-Type: application/json`, haciéndolo disponible como un objeto en la propiedad `req.body`.

urlencoded

Procesa una petición del tipo `application/x-www-form-urlencoded` generando el objeto `req.body` en la petición.

Este es el tipo MIME que se aplica por defecto a los datos que se envían a través de un formulario HTML, mediante peticiones tanto GET como POST. A estos datos se les aplican unas reglas de codificación especificadas en el estándar del W3C que son las que definen la estructura de una *query*, ya conocidas, como que los espacios se sustituyen por '+', los caracteres no alfanuméricos por su código ascii precedido de '%' de tal manera que tienen la forma %HH, o que los pares nombre/valor se separan con '&' de otros pares y con '=' entre ellos [52, Section 17].

multipart

Procesa los cuerpos de las peticiones multiparte, de tipo MIME `multipart/form-data`.

Este tipo se aplica a los datos que se envían cuando el tamaño de éstos es demasiado grande como para enviarlo como `application/x-www-form-urlencoded`. Esto sucede, por ejemplo, con archivos binarios donde para codificar cada octeto, hacen falta tres caracteres (%HH), lo cual triplica la cantidad de datos a enviar siendo altamente ineficiente.

En los mensajes multiparte, como su nombre indica, hay varias partes separadas con un limitador (una cadena de caracteres tal que no se encuentre en el cuerpo de ninguna de las partes que se transmite). Cada una de las partes tiene su juego de cabeceras, como su propio `Content-Type` o su `Content-Transfer-Encoding`. Destaca sobre ellas la cabecera `Content-Disposition` que, aparte de tener valor fijo (`form-data`) incluye el atributo name que da nombre a la parte [53].

cookieParser

Procesa el campo `Cookie` de la cabecera HTTP. De esta manera, podemos acceder a las *cookies* a través de las propiedades `req.cookies` o `req.signedCookies` que el *middleware* incluye en la petición `req`. Estas propiedades son un objeto cuyas claves son el nombre de la *cookie* y sus valores, el contenido de las mismas.

Como se ve, se puede activar el soporte para *cookies* firmadas pasando la clave al *middleware* como cadena de caracteres. Esto hará que la propiedad `req.secret` esté también disponible en `req`. Las *cookies* firmadas [54] son *cookies* a las que a su contenido se le añade un código comprobación en forma de *hash* criptográfico para evitar que se alteren en el cliente. El *hash* se calcula con el contenido del mensaje, mediante el algoritmo `sha256` y se codifica posteriormente en `base64`.

session

Implementa un completo sistema de gestión de sesiones las cuales residiran en memoria, por tanto, como se advirte, no apto para usar en producción.

Las sesiones son un mecanismo que se implementa en el lado del servidor para que la aplicación web mantenga cierta información relativa al cliente y

su interacción con ella. Ya que, como se recordará, HTTP es un protocolo sin estado y, por tanto, incapaz de cumplir esta función.

Con Connect, la información se estructura como un objeto JSON que se guarda en la petición como la propiedad `req.session`. En ella se pueden almacenar los datos que se quiera, añadiéndolos como propiedad suya, con la seguridad de que estarán disponibles para su uso en posteriores peticiones provinientes del mismo cliente.

Dependiente del middleware: *cookieParser*.

cookieSession

Habilita el mecanismo de sesiones pero guardando la información en la misma *cookie* en lugar de la memoria del servidor [55].

Dependiente del middleware: *cookieParser*.

methodOverride

Sobreescribe el método HTTP, accesible por `req.method`, de las peticiones que tengan presente el parámetro `_method` en la *query* o la cabecera `X-Http-Method-Override`. Esta cabecera es la manera que tienen los clientes de un servicio REST de realizar peticiones a este servicio sin que sean rechazadas por un *proxy*.

Se debe recordar que las operaciones CRUD (*Create-Read-Update-Delete*) en los servicios REST se basan en los métodos GET, POST, PUT y DELETE de HTTP. Los dos últimos consideran no seguros por los *proxies*, que pueden rechazar la petición respondiéndola con un error `405 Method Not Allowed`. La solución pasa por generar una petición GET o POST que incluya en la cabecera `X-Http-Method-Override` el verbo que realmente quiere emplearse (PUT o DELETE). Si la petición se realiza desde un formulario html, se puede incluir el campo oculto `_method` con el método a usar:

```
<input type="hidden" name="_method" value="put" />
```

Este *middleware* detectará la presencia de cualquiera de las dos soluciones y sustituirá el método de la petición por su valor, dejando el método original disponible en `req.originalMethod`.

responseTime

Añade a la respuesta la cabecera `X-Response-Time` indicando el tiempo que

ha tardado la aplicación en procesar la petición.

static

Aporta la capacidad de servir de manera eficiente fichero estáticos, como imágenes u hojas de estilo, por defecto contenidos en el directorio `./public`.

Gestiona automáticamente los tipos MIME de los ficheros y los errores que puedan surgir empleando los códigos de respuesta apropiados.

staticCache

Habilita una pequeña cache de archivos servidos por el *middleware static*. En total, mantiene en memoria los 128 archivos menores de 256Kb más solicitados al servidor, gestionando las cabeceras HTTP relacionadas con la caché. Ambas cifras son por defecto y pueden ser cambiadas a través de las opciones de *staticCache*.

No se aconseja su uso, puesto que estará deprecado a partir de la versión 3.0 de Connect.

directory

Genera un listado de los directorios y archivos contenidos en un directorio raíz dado, y lo envía en html, JSON o texto plano, según acepte el agente de usuario.

vhost

Facilita la creación de *hosts* virtuales basados en nombre [56].

En este caso, los *hosts* virtuales son aplicaciones web que corren en una misma máquina con una sola dirección IP. Se accede a cada *host* virtual creando un alias (`CNAME`) en el DNS apuntando a la máquina. El nombre del *host* es una de las cabeceras HTTP (`Host`) y es en lo que se basa este *middleware* para dirigir la petición.

favicon

El *favicon* es un icono que los navegadores solicitan al servidor de la página o aplicación web para identificarla en la barra de direcciones, pestaña del navegador y marcadores.

Este *middleware* ofrece el icono que está en el *path* que se pasa por parámetro. Si no se especifica, muestra uno por defecto.

Es útil porque el navegador realiza la petición `http://www.hostname.domain:`

port/favicon.ico automáticamente. Sin el *middleware* la petición llega al controlador de las rutas que puede direccionar mal la petición o no saber cómo manejarla, produciendo con toda probabilidad un error.

limit

Establece un límite para el cuerpo de las peticiones. El límite se pasa por parámetro en cifra o como una cadena de caracteres, soportando expresiones en `kb`, `mb` o `gb`, por ejemplo, '1.3mb'.

query

Únicamente hace disponible el objeto `req.query` como propiedad de la petición `req`. Hace uso de la librería *url* del core de Node y del módulo *qs* para procesar la *query*.

errorHandler

Adapta las trazas generadas por un Error al tipo MIME que el cliente solicite: html, json y text (por defecto).

Está pensado sólo para entornos de prueba (*test*), no para producción.

6.2. Express

Express, en palabras de sus creadores, es un *"framework de desarrollo de aplicaciones web minimalista y flexible para Node.js"* [1]. A pesar de lo que pueda pensarse del término minimalista, Express ofrece características muy potentes, entre las que caben destacar:

- robusto sistema de enrutamiento de peticiones
- soporte para generar páginas HTML dinámicas a partir de plantillas con capacidad de utilizar varios motores de renderizado de vistas

Construido encima de Connect, establece una fina capa de software que enriquece de manera sencilla, elegante y transparente las características que exponía el módulo HTTP, abstrayendo de él detalles muy concretos, simplificándolos y ampliando el rango de características manejables del protocolo HTTP al desarrollador.

[1] http://expressjs.com/

Para comenzar a usar Express, como todos los módulos de Node, debe solicitarse con

```
var express = require('express');
```

que importa el objeto factoría para crear aplicaciones Express, cuyo uso difiere ligeramente del que se ha visto anteriormente parar `require` (situar bajo un namespace las clases y funciones del módulo). Ahora, invocando esta función se genera directamente una aplicación web:

```
var app = express();
```

A partir de aquí se dispone de una aplicación de características avanzadas, como las descritas al inicio, que se ejecuta en un entorno, con sus correspondientes variables de entorno. Además, sirve como *proxy* de Connect, es decir, se puede hacer uso de las características de Connect invocando el método que corresponda a través de la variable que representa la aplicación Express, en este ejemplo, `app`.

Las variables del entorno determinan el comportamiento de varios aspectos de la aplicación, sirviendo como configuración de la misma. Se determina a través de ellas si el *script* está corriendo en el entorno de producción o de desarrollo, modo por defecto a menos que el programador o la variable de ejecución de Node `process.env.NODE_ENV` digan lo contrario.

Estas variables se consultan y modifican con:

- los métodos `accesores app.get(nombre)` y `app.set(nombre, valor)` para asignar y consultar los valores.
- los métodos `app.enable(nombre)` y `app.disable(nombre)` para habilitar o deshabilitar variables *booleanas*.
- los métodos `app.enabled(nombre)` y `app.disabled(nombre)` que se pueden emplear para consultar si se está utilizando o no la variable `nombre`.

Las variables que pueden modificarse con estos métodos corresponden a distintas características de la aplicación que se irán desgranando posteriormente. De una manera rápida, son las siguientes:

Ámbito	Variables

Generación de respuestas en notación JSON	`jsonp callback name`
	`json replacer`
	`json spaces`

Rutas de las peticiones según su URL	`case sensitive routing` `strict routing`

Sistema de generación de la parte de interfaz (vistas)	`view cache`
	`view engine`
	`views`

Tabla 6.1: Algunas de las variables configurables en los ajustes de Express

Ajustados los valores del entorno de la aplicación, se puede proceder a configurar la aplicación en sí. En una primera fase, debe determinarse la cadena de *middleware* que se quiere emplear en la aplicación. Este *middleware* son los componentes que provee Connect, ya que Express los pone a disposición del programador. También pueden ser funciones a medida de las necesidades del desarrollador, escritas por él mismo. La configuración se hará con el método `app.use([ruta], funcion)` que se invocará las veces necesarias hasta apilar todo el *middleware*. Es importante saber que la ejecución del *middleware* se realiza por orden de configuración: primero el que se configura antes. Como ejemplo:

```
app.use(express.favicon());
app.use(express.static(__dirname + '/public'));
app.use(express.cookieParser());
```

La siguiente fase pasa por configurar opcionalmente el motor de renderizado de plantillas para la generación dinámica de páginas HTML. Para ello:

- se configuran las variables de entorno relacionadas con el renderizado:
  - `views` es la ruta del directorio donde se alojan las plantillas que contienen el código de la vista. Por defecto es `'/views'`
  - `view engine` es el motor de renderizado, el encargado de procesar las plantillas y generar a su salida el código html
  - `view cache` habilita o deshabilita el cacheado de plantillas ya compiladas. Tener en caché una plantilla compilada acelera el renderizado de la página (por eso está activado en producción)

- se declaran las variables locales, que son aquellas comunes a todas las plantillas de la aplicación

- se configura qué motor de renderizado va a emplearse para las diferentes extensiones (en caso de que se usen varios) del recurso que se solicita. El método para ello es `app.engine(extension, funcionDeRenderizado)`

  La `funcionDeRenderizado` debe tener una cabecera concreta que admita los argumentos (ruta, opciones, callback) que Express espera que tenga para que funcione correctamente. De hecho, muchos motores tienen un método por convención que es `__express()`. Por ejemplo, el motor Jade:

  ```
  app.engine('jade', require('jade').__express); //CUIDADO: dos
                                                 //guiones bajos
  ```

  Como se ha comentado, Express es un *framework* muy flexible, por lo que admite varios motores, siempre que cumplan el requisito anterior. Por defecto se emplea Jade[2].

Por último, para tener la aplicación perfectamente funcional, deben definirse las rutas a las que un cliente puede acceder, con qué métodos HTTP puede hacerlo y el comportamiento que se espera de la aplicación. Para ello Express ofrece su sólido sistema de enrutamiento en el que cada método HTTP es una función de la aplicación cuyo primer argumento es la ruta sobre la que se realiza la petición (sin tener en cuenta la parte de la *query*). Esto significa que en la práctica existirán las funciones `app.get(ruta, [callbacks...], callback)`, `app.post(ruta, [callbacks...], callback)`, `app.put(ruta, [callbacks...], callback)`, `app.delete(ruta, [callbacks...], callback)`, etc...

El array de *callbacks* intermedios contiene funciones *middleware* que actuarán sobre la `ruta` en acciones muy concretas. El último argumento `callback` es el manejador de la `ruta` y contiene la lógica a ejecutar cuando se realiza una petición a ella.

Si una ruta admite todos los métodos entonces puede abreviarse con `app.all()`.

La potencia de Express en este ámbito radica en que las rutas pueden ser expresiones regulares. De hecho, una ruta identificada por una *String* se compila a una expresión regular. Además, se pueden parametrizar mediante *tokens*. Los

[2] http://jade-lang.com/

tokens representarán partes variables dentro de la estructura concreta de una URL. Con esto se da flexibilidad a la definición de la ruta y además se hace disponible esa parte a la lógica de la aplicación a través de la propiedad `params` del objeto *Request* que modela la petición. Un *token* se define con un identificador que comienza con dos puntos `':'`. Cuando llega una petición, Express analiza la URL y extrae de ella el valor del o los *tokens*. Ejemplo de ruta parametrizada puede ser:

```
/users/:uid/friends/:friendname
```

que abarcaría rutas como

`/users/34123/friends/arturo` o `/users/87345/friends/irene`.

Una vez definidas las rutas, éstas quedan reflejadas en el objeto `app.routes` desde donde se pueden consultar.

La función de *callback* que atiende las peticiones es similar a los *listeners* del módulo *http*: admite como argumentos la petición y la respuesta de la transacción. Express modela ambas como los objetos *Request* y *Response* que son una versión enriquecida de los objetos *ServerRequest* y *ServerResponse* del módulo *http* de Node, es decir, las propiedades de ambos serán las que tienen ambas clases más las que el distinto *middleware* que se ha configurado con Connect y el propio Express hayan añadido.

6.2.1. Request

Request es el modelo que Express proporciona para las peticiones HTTP. A través de él, el desarrollador tiene acceso de una manera sencilla a gran variedad de información que el cliente envía en las cabeceras, sin tener que incluir lógica adicional para procesarlas.*Request* proporciona soporte para las siguientes:

`Accept`

Express disecciona los tipos que el cliente acepta en la petición y construye `req.accepted`, un array donde están todos ordenados según prioridad del cliente. El desarrollador puede comprobar si el cliente acepta unos `tipos` determinados (separados por comas) con `req.accepts(tipos)`, que devuelve el tipo mejor soportado de todos.

`Accept-Charset`

Express disecciona esta cabecera y genera el array `req.acceptedCharset` con los juegos de caracteres que admite el cliente ordenados por preferencia. También ofrece la facilidad `req.acceptsCharset(charset)` para determinar si el cliente aceptaría un `charset` concreto.

`Accept-Language`
: al igual que en los dos casos anteriores, Express disecciona esta cabecera para obtener un array ordenado según las preferencias del cliente con los idiomas preferidos de éste. De manera análoga, con `req.acceptsLanguage(idioma)` se puede consultar si el cliente acepta un determinado `idioma`.

`Content-Type`
: de existir esta cabecera, con `req.is(tipo)` se puede comprobar si el cuerpo de la petición es ese `tipo`.

`Cookie`
: a través del *middleware* `cookieParser()` de Connect, Express disecciona las *cookies* de la petición y las almacena en `req.cookies` y en `req.signedCookies` en caso de que sean seguras.

`Host`
: la información que Express extrae de esta cabecera es el nombre del *host* `req.host` al que se dirige la petición, sin especificar el puerto en caso de ser éste distinto de 80. Además, todos los subdominios que haya en el `Host` se meten ordenados en `req.subdomains`

`Last-Modified` y `ETag`
: el cliente emplea estas cabeceras en peticiones que impliquen entidades que tenga cacheadas, llamadas *cache-conditional requests*. La RFC2616 [49, Section 13.3.4] establece las pautas que un cliente debe seguir para incluirlas en una petición. En base a éstas, Express determina si la entidad solicitada que el cliente posee es reciente, `req.fresh` es `true`, u obsoleta, con `req.stale`.

6.2.2. Response

Response es el modelo con el que Express modela las respuestas HTTP.

Como con el módulo *http* de Node, se puede establecer el *status code* de la respuesta, esta vez a través del método `res.status()` en lugar de modificar la propiedad `res.statusCode` de la clase *ServerResponse* del módulo *http*. Hay que recordar que modificar el *status code* sólo podía hacerse en los casos en que la respuesta tenía cabeceras implícitas, es decir, se había comenzado a generar la respuesta sin establecer explícitamente unas cabeceras. Además en ese caso y en este, sólo podía modificarse el *status code* si la respuesta no se había comenzado a enviar aún. Es por ello que el `res.status()` que ofrece Express es encadenable: a continuación de él se pueden invocar el resto de métodos de `res`.

Además, en el caso de tener que realizar redirecciones, Express ofrece facilidades con `res.redirect([responseCode], url)`, que redirecciona al cliente a la `url` (absoluta o relativa a la aplicación) con un código `302 Found` por defecto, pero que puede cambiarse con el argumento `responseCode`.

Algunas cabeceras se pueden manejar directamente a través de métodos específicamente hechos para ello:

`Content-Type`
: Express establece muchas veces el tipo MIME del contenido que se envía de manera automática pero el criterio del desarrollador prevalece si se emplea la función `res.type(tipo)`. Además, si la clase del tipo es texto (`text/*`), el juego de caracteres se puede especificar con la propiedad `res.charset`.

`Cookie`
: se puede enviar una *cookie* al cliente con `res.cookie(nombre, valor, [opciones])`

El resto de cabeceras se deben fijar con los métodos accesores `res.get(cabecera)` y `res.set(cabecera, valor)`. En este último caso se pueden fijar más de una si se pasa un único argumento con las cabeceras en notación JSON.

A la hora de enviar el cuerpo con la entidad solicitada, Express ofrece varias formas de hacerlo dependiendo de cómo se esté generando la respuesta. La manera más general es con `res.send([statusCode], cuerpoRespuesta)` que enviará respuestas que no sean de tipo *Stream*, es decir, ya están calculadas y se pueden enviar inmediatamente, por ejemplo, un objeto JSON o un *Buffer*. No entran en esta categoría archivos o páginas dinámicas.

Si lo que se pretende es enviar un archivo a través del *Stream*, Express proporcio-

na un método específico para ello: `res.sendfile(ruta, [opciones], [function(error){}])`. Este método envía el archivo ubicado en la `ruta` del sistema de ficheros (o relativa a `opciones.root`, si se especifica) fijando automáticamente el tipo según la extensión del mismo y su tiempo de caché si se indica en `opciones.maxAge`.

Para el caso de archivos html dinámicos, que necesitan ser procesados por algún motor de renderizado existe el método `res.render(vista, [locales], function(error, html){})`, al que se le indica como primer argumento el nombre de la vista.

La principal motivación de las páginas dinámicas es la de incluir trozos de código que se ejecuten durante el renderizado. Este código, en su ámbito de ejecución, puede hacer uso de variables compartidas con la lógica de la aplicación y cuyo valor se fija durante la ejecución de los métodos de la misma. Estas variables, las *locales*, serán visibles para la página dinámica siempre que se pasen como argumento a `res.render()` en un objeto JSON.

6.3. MongoDB

La base de datos no relacional MongoDB es uno de tantos proyectos ligados al movimiento NoSQL. Este movimiento, como su nombre indica, provee soluciones alternativas a las bases de datos donde se emplea SQL (*Structured Query Language*) como lenguaje de consulta. Con SQL se realizan peticiones a bases de datos donde la información se estructura en base a modelos de datos relacionales: las distintas entidades que representan esos modelos se almacenan en tablas entre las que se establecen relaciones. Una característica muy importante de estas bases de datos es cómo tratan las transacciones (grosso modo, conjuntos de operaciones) que se realizan sobre los datos. Éstas siguen un principio cuyo nombre indica cómo deben ser: ACID (*Atomic – Consistency – Isolate - Durability*). Según este principio, una transacción debe ejecutarse completamente o no hacerlo (Atomicidad), dejando la base de datos íntegra, no corrupta (Consistencia), sin interferir con otras transacciones (Aislamiento) y dejando los datos resultantes en un estado permanente (Durabilidad).

Sin embargo, el modelo de datos de NoSQL es mucho más flexible, existiendo varias formas de organizar la información: *clave-valor*, *clave-documento* o *BigTable*,

por citar algunas. Esta flexibilidad se ve reflejada también en la manera de realizar las consultas: ahora, en lugar de usar un lenguaje declarativo, perfectamente preciso, como SQL, se deja a la lógica de la aplicación realizar las operaciones sobre los datos directamente, con la intención de tener un mayor control sobre el rendimiento de la consulta.

Son precisamente características como éstas la que suponen sacrificar aspectos del principio ACID, por ejemplo, la Atomicidad, sólo garantizada a nivel de documento, o el Aislamiento, ya que cualquier dato leído por el cliente puede estar modificado por otro cliente concurrente [57]. Aún así, es cierto que algunos de los mecanismos se suelen poner a disposición del desarrollador para mitigar lo máximo posible esta carencia [58, Chapter 13].

Decantarse por el uso de NoSQL frente a SQL es una cuestión de análisis del problema: en general, se recomienda usar una base de datos no relacional cuando se tienen grandes cantidades de datos y/o se encuentran dificultades para modelarlos según un esquema. De otra manera, SQL es una tecnología muy madura con muchísimos años de investigación y desarrollo sobre ella [58].

MongoDB entra dentro de la categoría de bases de datos orientadas a documentos [59]: las entidades no son filas en una tabla sino un conjunto de pares *clave-valor* (campos), denominado documento, que a su vez se organiza en colecciones. Las consultas se realizan solicitando documentos cuyos campos cumplan uno una serie de criterios determinados.

Los autores del proyecto definen MongoDB como si fuera una base de datos relacional a la que únicamente se le ha cambiado el modelo de datos manteniendo otras características que tradicionalmente han funcionado muy bien, como índices o *updates*. Los puntos fuertes que destacan de MongoDB, según los autores [60], son:

flexibilidad, por usar como modelo de datos documentos en notación JavaScript (JSON). Estos documentos no tienen un esquema fijo, como sucede con los documentos XML, más rígidos en el sentido que los elementos deben ajustarse al esquema establecido para ellos, y como sucede en las bases de datos relacionales, donde un elemento a insertar en una tabla debe cumplir las condiciones fijadas para las columnas de dicha tabla.

potencia, por incorporar mecanismos propios de bases de datos relacionales que se suman al resto de ventajas. Entre ellos, los autores destacan los índices

secundarios, consultas dinámicas, ordenación o operaciones *"upsert"* (*update* si el documento existe, *insert* si no)

velocidad, porque agrupa toda la información realcionada en documentos en lugar de en tablas como ocurre en las bases de datos relacionales

escalabilidad horizontal, es decir, posee mecanismos para que sea muy fácil y seguro, sin interrupciones de servicio, añadir más máquinas corriendo MongoDB al *cluster*

facilidad de uso, por la sencillez con la que se instala y configura MongoDB, teniendo una instancia corriendo en muy pocos pasos.

Los criterios seguidos en el proyecto para elegir MongoDB serán el primero y el último de los puntos anteriores ya que, de momento, no se busca gran potencia ni escalabilidad. Además sirve de introducción a una tecnología que, a pesar de sus detractores, sigue creciendo y ganando popularidad, habiéndose adoptado en sitios y aplicaciones web de gran popularidad como Craiglist[3] o Foursquare[4].

La integración entre MongoDB y Node se hace a través de Mongoose[5], uno de los *drivers* de la base de datos para esta plataforma. Mongoose ofrece al desarrollador una forma sencilla de modelar los objetos de su aplicación y realizar sobre ellos las operaciones que permite MongoDB.

Usar Mongoose en una aplicación web es bastante fácil y sólo requiere un par de pasos:

- instalar el *driver*, por supuesto habiendo instalado ya Node y Express, con

  ```
  npm install mongoose
  ```

- importar el módulo en el código de la aplicación de la manera habitual

  ```
  var mongoose = require('mongoose');
  ```

- conseguir una conexión con el servidor MongoDB

  ```
  mongoose.connect(IPServirdor, nombreBaseDeDatos);
  mongoose.on('error', function(error){});
  ```

[3] http://www.craiglist.org
[4] https://foursquare.com
[5] http://mongoosejs.com/

A partir de este punto, si no se ha obtenido ningún error, el desarrollador será capaz de crear objetos a partir de un modelo. Este modelo sigue un esquema con el que se guardarán sus instancias en la base de datos. Un esquema puede verse como la definición de una colección ya que define la forma que tendrán los documentos de dicha colección.

Todo lo anterior no implica que se incurra en una contradicción con la característica *schemaless* (sin esquema) que MongoDB propone, sino que un esquema es una manera de estructurar los datos frente a la libertad que MongoDB ofrece para hacerlo o no. Lógicamente, en la solución a un problema real es deseable uniformidad en el modelo de datos. Aún así, estos esquemas se pueden modificar de una manera fácil y rápida.

Por tanto, el último paso antes de realizar operaciones con la base de datos es:

```
// Generar el esquema de una coleccion
var miEsquema = mongoose.Schema({miPropiedad: tipoDeDato});
// Generar el modelo del documento como un constructor Javascript
var MiModelo = mongoose.Model('MiModelo', miEsquema);
```

Por supuesto, este patrón de código se utilizará tantas veces como sea necesario para hacer uso de todos los modelos que se hayan estimado necesarios en la arquitectura de la aplicación.

En este momento la lógica de la aplicación puede hacer uso de los documentos de la colección instanciándolos de la manera clásica:

```
var miModelo = new MiModelo({miPropiedad: 'valor'});
```

La instancia creada posee métodos que interaccionan con MongoDB. Los más comunes son:

`miModelo.save([function(error, miModelo){}])`
: con el que se guardará el documento `miModelo` en la base de datos, tanto como si es de reciente creación como si se está realizando una actualización (*update*). Emplear este método es más usual si previamente se ha realizado una búsqueda o se acaba de instaciar el objeto, ya que en caso de querer actualizarlo se recomienda utilizar el método `MiModelo.update()` sobre el modelo del documento, no sobre la instancia en sí.

`miModelo.remove([function(error, miModelo){}])`
: con el que se borra el documento de la base de datos. Aún resultando con éxito, el *callback* opcional recibe una copia del objeto recién eliminado como argumento.

Asimismo, algo fundamental es la localización de documentos a través de consultas (*queries*), una operación que se hace a través del modelo del documento en lugar de la instancia del mismo, a diferencia de los casos anteriores:

```
MiModelo.find(query,
              [condiciones],
              [opciones],
              [function(err, resultSet){}])
```

o si sólo se quiere encontrar un documento:

```
MiModelo.findOne(query,
                 [campos],
                 [opciones],
                 [function(err, resultSet){}])
```

El argumento `query` contiene las condiciones que deben cumplir los documentos para formar parte del resultado de la búsqueda. En concreto se refieren a valores de uno o varios campos de éste.

El argumento `campos` es una *String* opcional con el nombre de los campos del documento, separados por un espacio, que se quieren obtener si es lo que se desea en lugar de obtener el documento completo.

Las `opciones` por su parte no están relacionadas con la finalidad de la búsqueda en sí sino con el resultado de ella, por ejemplo, limitarla a un número determinado de resultados o ordenarlos con algún criterio.

Un ejemplo de *query* sobre un documento que representa a un usuario, podría ser:

```
UserModel.find(
  {username : /^a\w+/, age: {$gt: 30}, accountType: 'premium'},
  'username email',
  function(error, resultSet){}
);
```

que devolverá como resultado un conjunto `resultSet` de nombres de usuario y su `email` asociado de cuentas `'premium'` de usuarios cuyo nombre comience por la letra *'a'* y tengan edades superiores a 30 años.

La función de *callback* del cuarto argumento también es opcional y si no está presente, el método `find()` sólo definiría la consulta, que se ejecutaría más tarde con el método `exec()`:

```
var consulta = miModelo.find({});
// Lineas de codigo adicionales
consulta.exec(function(error, resultSet){}));
```

Las funciones relativas a búsqueda son encadenables, es decir, pueden concatenarse con funciones que filtren el resultado, de tal manera que la consulta anterior podría escribirse también así:

```
UserModel.find(
  {username : /^a\w+/, accountType: 'premium'},
  'username email')
.where('age')
.gt(30);
```

Esta mínima introducción a Mongoose debería ser suficiente para comprender la dinámica de uso de la base de datos. El API del *driver* es extenso y muy flexible pero muy sencillo de utilizar, por lo que se recomienda su lectura en los casos en que las consultas y operaciones requieran mayor detalle.

6.4. Aplicación de ejemplo

Se va a realizar una versión reducida de la popular red de *micro-blogging* Twitter a la que se denominará Whizr. Así, un usuario de Whizr será un *whizr* y las publicaciones que haga en su perfil serán los *whizs*. A través de los tres objetivos propuestos se pretende alcanzar un grado de funcionalidad tal que la haga utilizable.

Como paso previo, debe tenerse una instancia de MongoDB corriendo en la máquina. En ella se creará la base de datos, a la que se llamará `whizr`, que empleará la aplicación. Para generarla, basta con conectarse con el cliente `mongo` a MongoDB e introducir a través del *prompt* el comando `use whizr`.

Aunque no sería necesario, ya que MongoDB lo haría implícitamente, se crearán las collecciones que se usarán a lo largo del desarrollo. Una contendrá los usuarios, los *whizrs*, y otra las publicaciones, los *whizs*. Los dos comandos necesarios para ello son:

```
db.createCollection("whizrs");
db.createCollection("whizs");
```

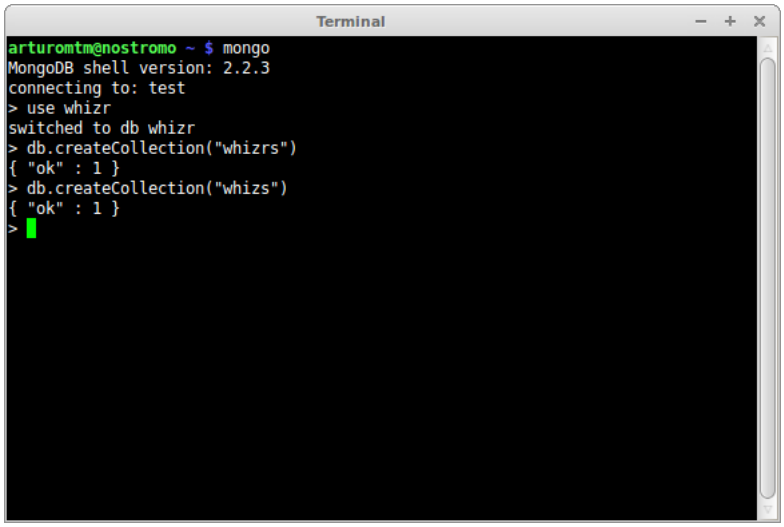

Figura 6.1: Creación de las Colecciones en el *prompt* de MongoDB

6.4.1. Clon de Twitter, objetivo 1: autenticación y control de sesiones

6.4.1.1. Descripción del Objetivo

El primer objetivo que se debe alcanzar a la hora de crear una red social es aquel que permite la propia existencia de la red: la posibilidad para un usuario de darse de alta y acceder a ella.

Por tanto, lo que se pretende alcanzar con este caso de uso es que el usuario:

- sea capaz de registrarse o *hacer login* en la aplicación, a través de la página de inicio
- acceda a la página de su perfil, una vez registrado o satisfactoriamente autenticado

- tenga la posibilidad de terminar su sesión, una vez haya accedido a su perfil

Estas acciones implican la existencia de:

- una página principal cuya ruta por defecto sería `'/'`.
- una página de perfil de usuario cuya ruta sería, por ejemplo, *'/nombreDeUsuario'*
- tres *endpoints* que atiendan las peticiones de crear una cuenta y hacer login y logout. En este caso determinados por las acciones `'/register'`, `'login'` y `'/logout'`

Hay que tener en cuenta que el acceso a los *endpoints*, éstos descritos y los futuros, debe ser restringidos en función de qué derechos posea el usuario sobre los mismos, es decir, qué acciones puede o no realizar en función de si está logueado o no o si es el propietario de la cuenta. Para ello se necesita hacer uso de un mecanismo de control de sesiones (creación, mantenimiento y destrucción) y la lógica necesaria para proteger dichos *endpoints* en base a ese mecanismo.

Por último, se debe articular un modelo de datos que permita manejar los usuarios en este sencillo caso de uso.

6.4.1.2. Diseño propuesto al Objetivo implementado con Express

Express proporciona un script, *express*, para el *scaffolding* que desde línea de comandos se encarga de generar la estructura de directorios de la aplicación. Esta estructura es muy sencilla, se compone de tres directorios: `public`, para los recursos estáticos como hojas de estilo o imágenes, `routes`, con la lógica de aplicación que se defina en cada ruta, y `views` que contine las plantillas con las vistas que se renderizarán desde las rutas.

Las opciones del comando *express* son limitadas pero suficientes para conseguir el máximo grado de funcionalidad en una aplicación web. Para esta práctica serían útiles las opciones `-e`, que añade soporte para el motor de renderizado *ejs*, y `-s`, que añade soporte para sesiones.

En definitiva, y como se acaba de comentar, *express* generará el *scaffolding* que indica su salida:

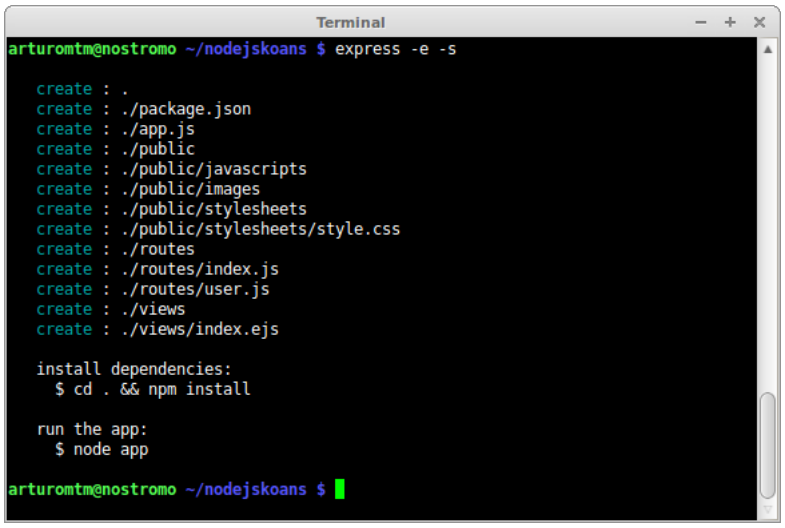

Figura 6.2: *Scaffolding* creado por Express

Para esta aplicación no se empleará la estructura generada por *express*, ya que es muy sencilla, aunque será una versión ligeramente modificada, por simplicidad, y respetando al máximo las convenciones seguidas por el comando. Para aplicaciones más complejas se recomienda el uso del comando.

Como es habitual en las aplicaciones para Node, se empieza importando los módulos que se van a emplear y, como paso siguiente, se crea una instancia del servidor *Express*:

```
var express = require('express'),
    ejs = require('ejs');

var app = express();
```

Antes de empezar a escribir la lógica de la aplicación se debe configurar el *middleware* que se va a utilizar teniendo en cuenta los requisitos. Éstos son:

- servir contenido estático: la página de inicio, la hoja de estilo y código javascript para el lado del cliente
- necesidad de una capa de manejo de sesiones
- facilidades para procesar los parámetros de los formularios en peticiones `POST`. Se recuerda que el `Content-Type` de este tipo de peticiones es `application/x-www-form-urlencoded`

- a priori puede no ocurrírse la necesidad, pero los navegadores actuales solicitan el icono de favoritos, `favicon.ico`, a la aplicación lo que interferirá sin duda al controlador de las rutas y con mucha probabilidad causar un error, así que se incluirá esta capa en la pila *middleware*

Teniendo esto en cuenta, la configuración que queda es:

```
app.configure(function(){
  app.use(express.favicon());
  app.use(express.static(__dirname + '/public'));
  app.use(express.cookieParser());
  app.use(express.session({ secret: "Node.js koans" }));
  app.use(express.bodyParser());
});
```

El siguiente paso ahora es definir las rutas y las acciones a realizar cuando se invoquen pero antes, y puesto que las acciones involucran trabajar con el modelo de datos de los usuarios de la aplicación, debe definirse este modelo. Se hará en un módulo aparte que se importará en el código principal de la aplicación, sin capas intermedias que abstraigan la base de datos por dos pequeños motivos: uno, mantener el foco sobre Node y, dos, para no introducir más elementos en la arquitectura de una aplicación que a priori se antoja muy sencilla.

```
var whizrSchema = new Schema({
  name        :  String,
  username    :  String,
  password    :  String,
  email       :  String,
  emailHash   :  String
});
```

Los campos que componen el modelo son los que se aceptan en el formulario de registro más uno adicional que contiene el *hash MD5* de la dirección de correo normalizada en minúsculas. El motivo es meramente estético: ese *hash* sirve para solicitar al API del servicio de fotos de perfil, o avatares, Gravatar[6] una imagen de perfil para el usuario en caso de tener cuenta en dicho servicio. Esta manera de obtener automáticamente una imagen de perfil introduce una ruta adicional auxiliar `/username/profilepic` que atenderá las peticiones de las imágenes,

[6] http://www.gravatar.com

redirigiéndolas a la URL de Gravatar.

Volviendo a las rutas, como se ha descrito anteriormente, se necesita una página principal, servida desde la ruta base `'/'`, que será estática puesto que su objetivo es presentarnos un par de formularios:

- uno para registrarse, proveyendo los campos `username`, `password`, `email` y `nombre` y `apellidos`
- y otro para acceder a la cuenta de usuario, con la identificción basada en el `username` y la `password`

Estos formularios envían los datos con el método `POST` a los *endpoints* `'/register'` y `'/login'`. Ambos aceptan parámetros y son rutas de acceso general (no restringido únicamente a usuarios logueados en el sistema) y se encargan de comprobar que los campos cumplen las características que se les exigen redirigiendo al usuario donde proceda: en caso de éxito, a la página del perfil de usuario, y en caso de error, `'/register'` envía un código HTTP `400 Bad Request` y `'/login'` un código HTTP `404 Not Found`.

Para la acción de registro `'/register'` los campos serán válidos si el nombre de usuario, `username`, no está repetido en la base de datos y no excede los 10 caracteres, la `password` tiene entre 5 y 10 caracteres, y la dirección de correo es una dirección válida, es decir, contiene un símbolo @ y un dominio. El nombre con los apellidos tampoco deberán superar los 15 caracteres.

Para la acción de acceder `'/login'`, el nombre de usuario debe estar contenido en la base de datos y además la `password` asociada a él debe ser también la que esté en la base de datos.

Cuando las condiciones impuestas en la ejecución de las acciones anteriores se verifican correctamente, se genera una sesión donde se almacenarán en una variable los datos del usuario para posterior acceso. Chequear la existencia de esta variable y su contenido servirá para saber si una sesión es válida y quién es el usuario que la ha iniciado, útil para determinar qué permisos tiene.

```
app.post('/register', function(req, res){
  var username = req.param('username');
  var password = req.param('password');
  var name = req.param('name');
  var email = req.param('email');
```

```
  if ( username.length <= 0 ||
       username.length >= 10 ||
       password.length <= 0 ||
       password.length >= 10) {
         res.redirect(400); // error
  }

  models.Whizr.findOne({ username: req.params.username },
   function(err, doc){

    if (err) {
      res.send('Error', 500);
    }

    if (doc != null) {
      res.send('Error', 404);
    }

    var whizr = new models.Whizr();
    whizr.name = name;
    whizr.username = username;
    whizr.password = password;
    whizr.email = email;
    var hash = crypto.createHash('md5');
    hash.update(email);
    whizr.emailHash = hash.digest('hex');

    whizr.save(function(err){
      if (err) {
        res.send('Error', 500);
      }
      res.redirect('/' + username);
    });
  });
});
```

```
app.post('/login', function(req, res){
  // En caso de estar logueados
  if (req.session.whizr != undefined) {
    res.redirect('/' + req.session.whizr.username);
  }
  models.Whizr.findOne( { username: req.param('username') },
    function(err, doc){
      if (err) {
        res.send('Error', 500);
      }

      if ( doc.password == req.param('password') ) {
        req.session.whizr = doc;
        res.redirect('/' + doc.username);
      } else {
        res.send('Unauthorized', 401);
      };
    })
});
```

Una vez establecida la sesión, se realiza una redirección `302` a una página dinámica, servida en la ruta `'/nombreDeUsuario'` y generada con el motor de renderizado *ejs*. Esta página muestra la información básica de la cuenta del usuario y ofrece la posibilidad de terminar la sesión con la acción *logout*.

La ruta `POST` restante, `'/logout'`, no recibe parámetro alguno y tiene restringida su invocación únicamente a usuarios logueados en el sistema. Un resultado satisfactorio de la acción redirige al usuario a su perfil por el método tradicional, un código `302`, y un resultado fallido genera un error.

Como realizar peticiones a determinadas rutas de la aplicación debe protegerse, como es este caso, se ha optado por hacer uso de una función *middleware* que verifique que exista una sesión activa para el agente de usuario, o lo que es lo mismo, que en la sesión exista el objeto `whizr`, creado cuando el proceso de login se realiza satisfactoriamente. De no existir sesión alguna, el usuario recibirá el error apropiado para estos casos: `401 Unauthorized`.

La filosofía de la función es la misma que cuando se diseña un nuevo *middleware* para Connect: una función que recibe en sus parámetros los objetos petición,

`req`, y respuesta, `res`, y también el siguiente *middleware* o función a ejecutar. Por tanto, se invocará antes de la función que procesa la petición.

```
var checkAuth = function(req, res, next){
  req.session.whizr?
    next() :
    res.send('Unauthorized', 401);
}
```

Se añade al *stack middleware* pasándola por parámetro al manejador de la ruta `post()`:

```
app.post('/logout', checkAuth, function(req, res){
  var redirect = req.session.whizr.username;
  req.session.destroy();
  res.redirect('/' + redirect);
});
```

Si el proceso de registro o autenticación ha sido correcto, la redirección se hace a la URL del perfil de usuario que presenta la página del perfil generada dinámicamente. Por ello es imprescindible configurar:

```
app.register('.html', require('ejs'));
app.set('view engine', 'ejs');
app.set('view options', { layout: false });
```

Esta URL es una ruta parametrizada mediante el token `:username` que Express resuelve dinámicamente aplicando la lógica que se ha programado para ella.

El último detalle del caso de uso es el comportamiento de la aplicación cuando se accede, mediante el método GET, a la URI *'/nombreDeUsuario'* sin que el cliente tenga establecida una sesión activa. Aquí, la aplicación debe simplemente mostrar el perfil asociado a dicho nombre de usuario, y, de nuevo, la posibilidad de registrarse o hacer login en la aplicación. No se ofrecerá ninguna opción más excepto las que se vayan introduciendo en posteriores casos de uso.

```
app.get('/:username', function(req, res){

  var username = req.params.username;

  models.Whizr.findOne({ username: username }, function(err, doc){
```

```
    if (err || doc == null) {
      res.send('Not found', 404);
    };
    res.render('home.html', { name: doc.name,
                              username: doc.username,
                              whizr: req.session.whizr });
  });
});
```

Por último, se habilita la aplicación para que acepte peticiones en el puerto (arbitrario) `8080`:

```
app.listen(8080);
```

Se puede apuntar el navegador, en el caso de que se esté sirviendo la página desde la misma máquina, a la url `http://localhost:8080` y ver el resultado:

Figura 6.3: Página principal de la aplicación con Express

6.4.1.3. Objetivos de los Koans

La primera parte de los Koans de este capitulo cubre aspectos relacionados con la configuración y los fundamentos más básicos del desarrollo de una aplicación con Express.

1. Configurar una aplicación `app` Express indicando el uso de determinados componentes *middleware* como `favicon`:

```
app.configure(function(){
  app.___(express.favicon());
  // Resto de configuracion
});
```

2. Habilitar a la aplicación `app` para que sea capaz de atender peticiones realizadas con el método HTTP `GET` en una determinada ruta con `app.get()`:

```
app.___('/', function(req, res){
  // Logica del endpoint
});
```

3. Enviar respuestas que no sean resultado de peticiones a recursos estáticos ni contenido generado a través de un *Stream*, sino simplemente una respuesta sencilla, a través de `send()`:

```
if (err || doc == null) {
  res.___(404, 'Not found');
}
```

4. Devolver contenido html dinámico a las peticiones que lo soliciten, empleando el motor de renderizado definido para la aplicación a través del método `render()`:

```
res.___('home.html', { name: doc.name,
                       username: doc.username,
                       whizr: req.session.whizr } )
```

5. Habilitar a la aplicación `app` para atender correctamente peticiones realizadas con el método HTTP `POST` en la ruta que se defina previamente con `app.post()`

```
app.___('/login', function(req, res){
```

```
    // Codigo con la logica del endpoint
});
```

6.4.1.4. Preparación del entorno y ejecución de los Koans

Siguiendo el mismo procedimiento que en las anteriores prácticas, es necesario importar el módulo *koanizer*, proporcionado en las dependencias, para poder ejecutar los Koans de manera satisfactoria.

El fichero que contiene los *koans* es `express-koans.js`, que lo único que exporta es la propia aplicación Express, lista para ponerse a escuchar. De la misma manera que en prácticas anteriores este fichero debe ser editado y completado, sin que quede ningún hueco ___.

Para verificar que se han realizado con éxito los *koans* se ejecutan los casos de prueba con:

```
$ jasmine-node -verbose spec/
```

La correcta ejecución de las anteriores pruebas resulta en una aplicación totalmente funcional y ejecutable mediante el comando:

```
$ node app.js
```

que únicamente pone la aplicación Express a escuchar en el puerto 8080:

```
require('./express-koans').listen(8080);
```

Para realizar operaciones sobre la lista, se debe apuntar un navegador a la URL `http://localhost:8080`, en caso de que se esté ejecutando en local. A partir de este punto el resto está suficientemente explicado a lo largo de los apartados anteriores.

Cabe indicar que es completamente necesario que exista una instancia de MongoDB corriendo en el sistema. Por lo general, en entornos Unix, se puede instalar como un *daemon* o servicio que arranca al inicio del sistema. De no ser así, la base de datos siempre puede levantarse con el comando `mongod`, en principio sin parámetros si no se ha tocado la configuración que por defecto ofrece:

```
$ mongod
```

En caso contrario, lo más común es que la instalación haya alterado la ruta al directorio donde se almacenan los datos de la base de datos. Por defecto ésta es `/data/db` pero podría localizarse en `/var/lib/mongodb` de tal manera que ahora el comando sería:

```
$ mongod –dbpath /var/lib/mongodb
```

6.4.2. Clon de Twitter, objetivo 2: publicación de whizs y follow y unfollow de otros usuarios

6.4.2.1. Descripción del Objetivo

El segundo objetivo que se debe alcanzar a la hora de crear una red social es, precisamente, el que da sentido al concepto: otorgar a los usuarios la posibilidad de crear relaciones entre ellos y de generar contenidos.

Lo que ahora se pretende es que el usuario:

- pueda publicar un *whiz* y que éste aparezca en su página de perfil
- tenga la posibilidad de, cuando visite el perfil de otro usuario, indicar a la aplicación que comienza o deja de seguirlo con vistas a futuras interacciones entre ellos.

Esto, básicamente, introduce tres nuevos endpoints que respondan a las acciones `'/whiz'`, `'/follow'` y `'/unfollow'`. Es obvio que el acceso a estas rutas sólo debería estar permitido a usuarios autenticados en el sistema, por lo que se hará uso de los mecanismos ya implementados para protegerlas.

6.4.2.2. Diseño propuesto al Objetivo implementado con Express

La introducción de tres nuevas acciones implica, además de las funciones respectivas en el servidor para su procesamiento, la inclusión de tres nuevos formularios en el lado del cliente con los que enviar mediante `POST` los datos que las nuevas funcionalidades requieran:

- Un formulario sencillo que envíe los *whizs* al servidor. Bastará con un campo de longitud de texto máxima de 140 caracteres y un botón para enviar.

- Dos formularios que constan únicamente de un botón, que envíen al servidor el nombre de usuario de la persona a la que se desea seguir o dejar de seguir. Estos formularios deben ser excluyentes, es decir, si no se sigue al usuario, en su perfil debe aparecer el formulario *'Follow'*, y si se sigue, el formulario *'Unfollow'*.

Como los *whizs* aparecen en la página de perfil del usuario, aquella definida por la ruta *'/nombreDeUsuario'*, no es necesario añadir nuevos elementos de presentación html, ni estáticos ni dinámicos. Bastará con modificar los existentes.

En base a esto, los formularios se pueden incluir en la página del perfil que es, de hecho, el lugar idóneo para hacerlo. Las restricciones que se encuentran son:

- que sólo aparezcan cuando el usuario está logueado. Obviamente, un usuario no autenticado no debe ser capaz de publicar un *whiz* ni seguir o dejar de seguir a otro.
- que un usuario no pueda seguirse a sí mismo. Se deben evitar situaciones absurdas.

Los datos que envían cada uno de ellos servirán para actualizar la base de datos, con lo que el modelo de datos de la aplicación deberá ser extendido y actualizado. Se puede introducir el esquema de una nueva entidad que represente a un *Whiz*:

```
var whizSchema = new Schema({
  text     :  { type:String },
  author   :  { type:String },
  date     :  { type:Date, default: Date.now }
});
```

Así mismo, se debe reflejar que el usuario posee una lista de los otros usuarios a los que sigue con lo que se hace necesaria la extensión del esquema *Whizr* mediante un nuevo campo, *following*, donde se enumeren los nombres de usuario de aquellos *Whizrs* a los que se sigue:

```
var whizrSchema = new Schema({
  name      :  String,
  username  :  String,
  password  :  String,
```

```
  email     :  String,
  emailHash :  String,
  following :  [String]
});
```

Los nuevos endpoints serán los encargados de actualizar la base de datos según su cometido. Los resultados de la acción que debe esperar el usuario son:

- si ha invocado `'/whizr'`, redirección a la página de su perfil en caso de éxito al guardar en la base de datos el texto del *whiz* con una referencia al autor. En caso de que el *whiz* no pueda publicarse, se genera un error de servidor `500 Internal Error`.

```
app.post('/whizr', checkAuth, function(req, res){
  var text = req.param('whiz');
  if ( text == null || text.length  == 0 || text.length >= 140) {
    res.send(400, 'Error');
  }
  var whiz = new models.Whiz();
  whiz.text = text;
  whiz.author = req.session.whizr.username;
  whiz.save(function(err){
    if (err) {
      res.send(500, 'Error');
    }
    res.redirect('/' + req.session.whizr.username);
  });
});
```

- si ha hecho `'/follow'` o `'/unfollow'` satisfactoriamente, habrá redirección a la página del perfil del usuario al que ha empezado o dejado de seguir. Si fracasa, por ejemplo porque no se suministre el nombre de usuario, se debe obtener un error `404 Not Found`.

En el caso concreto de `'/unfollow'`, no sólo es necesario una actualización de la base de datos, sino también de la información sobre usuarios seguidos que existe en la sesión con objeto de evitar inconsistencias. Se sabe que en la sesión está replicada toda la información del perfil del usuario contenida en la base de datos pero en lugar de refrescarla desde ella, se hará al vuelo desde el mismo *callback*

del método que la actualiza:

```
models.Whizr.update(
  {username: whizr },
  { $pull: { following: unfollow } },
  null,
  function(err, numAffected){
    if (!err) {
      // actualiza la sesion evitando realizar
      // una peticion a la base de datos
      var following = req.session.whizr.following;
      following.splice(following.indexOf(unfollow), 1);
      res.redirect('/' + unfollow);
    }
});
```

6.4.2.3. Objetivos de los Koans

La segunda parte parte de los Koans de este capitulo está referida a conceptos de nivel medio y más orientados a la parte de base de datos:

1. Comprender la mecánica de una función *middleware* y su ciclo de vida, en concreto, el momento de delegar en el siguiente *middleware* invocándolo con `next()`:

   ```
   var checkAuth = function(req, res, next){
     req.session.whizr? ___() : res.send('Unauthorized', 401);
   }
   ```

2. Aplicar el filtro *middleware* anterior a las rutas en que tenga sentido. En este caso, sobre los *endpoints* que necesiten autorización para invocarse, como `logout`, que se modifica respecto a su versión en el Objetivo#1:

   ```
   app.post('/logout', ___, function(req, res){
     var redirect = req.session.whizr.username;
     req.session.destroy();
     res.redirect('/' + redirect);
   });
   ```

3. Actualizar elementos de la base de datos con el método `update()` que ofrecen los modelos definidos con Mongoose:

```
models.Whizr.___( {username: req.session.whizr.username},
                  {$addToSet: { following: followTo }},
                  null,
                  function(err, numAffected){
                    //Acciones post-actualizacion
                  }
                );
```

4. Emplear junto con el método `update()` alguno de la gran cantidad de operadores que están disponibles en MongoDB y que también son utilizables a través de Mongoose, como `$pull`:

```
models.Whizr.update( {username: req.session.whizr.username },
                     {___: { following: unfollow } },
                     null,
                     function(err, numAffected){
                       //Acciones post-actualizacion
                     }
                   );
```

6.5. Conclusión

Con la finalización de este capítulo se debe haber conseguido la capacidad de usar y configurar cualquier *middleware* que Connect incluye. A partir de ellos se podrá empezar a construir un aplicación web completa con Express, que atienda con `get()` o `post()` a peticiones hechas con los distintos verbos HTTP sobre aquellas rutas que se parametricen para ellos.

Capítulo 7

Socket.IO

7.1. ¿Qué es Socket.IO?

Una de las características más relevantes de Node, como se ha señalado en la Introducción a la Plataforma, es su idoneidad para aplicaciones en tiempo real gracias a su eficiencia en el procesamiento de conexiones masivas. Ejemplos de este tipo de aplicaciones son los chats o los juegos multijugador.

La librería más famosa, estándar de facto en Node, para este tipo de aplicaciones es Socket.IO. En palabras de sus creadores, *"Socket.IO pretende hacer posible las aplicaciones en tiempo real en cada navegador y dispositivo móvil, difuminando las diferencias entre los diferentes mecanismos de transporte"* [61].

En otras palabras, Socket.IO es una capa intermedia en la comunicación Cliente-Servidor que abstrae el mecanismo de transporte (la manera en que se realiza la conexión) de dicha comunicación, empleando en el cliente el mejor soportado y más adecuado de entre todos los posibles. Consecuentemente, Socket.IO no es sólo una librería para Node sino que lo es también para la parte cliente.

7.1.1. "Socket.IO pretende hacer posible las aplicaciones en tiempo real"

Socket.IO no sólo ofrece un adaptador de capa de transporte sino que añade más características propias de las aplicaciones en tiempo real como *heartbeats*, *timeouts* y *disconnection support* [62].

Estas tres características son parte del mecanismo del control del estado y recuperación de la conexión para que, en caso de pérdida de la misma, se pueda re-establecer. Una por una:

- Los *heartbeats* son mensajes que se intercambian cliente y servidor en intervalos periódicos de tiempo negociados. El propósito es detectar cuándo una conexión se ha interrumpido. En el caso de Socket.IO son una característica del nivel de transporte y su funcionamiento es muy sencillo: cuando el servidor recibe un mensaje *heartbeat*, espera un tiempo `heartbeatInterval` para enviar otro y vuelve a esperar durante un tiempo `heartbeatTimeout` en recibir otro *heartbeat*, con lo que repite el proceso. Si pasado ese tiempo no hay respuesta por parte del cliente, da la conexión por perdida.

  Por su parte, el cliente también espera un tiempo `heartbeatTimeout` cada vez que recibe un paquete, independientemente si es de tipo heartbeat o no, para dar la conexión por cerrada.

- Los *timeouts*, como, por ejemplo, el `heartbeatTimeout`, son el límite de tiempo máximo, negociado entre el cliente y el servidor, que debe transcurrir antes de dar una conexión como perdida.

- El soporte a la desconexión es la capacidad que tiene el objeto *Socket* del cliente Socket.IO cuando detecta, a través del mecanismo de *heartbeats*, que la conexión entre cliente y servidor se ha interrumpido y, como respuesta, comienza a realizar intentos de reconexión con el servidor. Estos intentos se realizan periódicamente en intervalos de tiempo de duración fijada opcionalmente por el cliente, hasta un máximo de intentos que puede ser infinito. El tiempo *reconnectionDelay* entre intervalos se duplica en cada intento de reconexión según un algoritmo *exponential back off*.

Estas características dan soporte a un API orientada al intercambio de eventos entre las dos partes.

7.1.2. "en cada navegador y dispositivo móvil"

La intención de esta librería es ofrecer un soporte universal de comunicación en tiempo real, sea cual sea el agente de usuario que se utilice con las características que cada uno de ellos ofrece. Hasta el momento de la elaboración de este capítulo, versión 0.9, se soportan los navegadores Internet Explorer 5.5+, Safai 3+, Google

Chrome 4+, Firefox 3+ y Opera 10.61+ para escritorio, e iPhone Safari, iPad Safari, Android Webkit y WebOS Webkit para móviles.

7.1.3. "difuminando las diferencias entre los diferentes mecanismos de transporte"

La parte de cliente de Socket.IO, cuando se conecta, decide qué modo de transporte es el que va a utilizar de todos los disponibles o de aquellos que el usuario elija opcionalmente. Se entiende por modo de transporte a la tecnología o a la técnica empleada para realizar la conexión Cliente-Servidor según esté soportada por el agente de usuario. Hay una lista bastante amplia que cubre los distintos tipos de agente de usuario (viejos navegadores, navegadores para móviles...). Los diferentes mecanismos de transporte, ordenados según preferencia de utilización por la librería, que Socket.IO soporta son:

Websocket

Bajo el nombre de *Websocket* se engloba tanto el protocolo *Websocket*, definido en la RFC6455 [63], como el API que permite a las páginas web el uso del mismo [64].

Este protocolo básicamente habilita en el agente de usuario, no necesariamente el contexto exclusivo, una comunicación *full-duplex* Cliente-Servidor sobre una única conexión TCP. La motivación del protocolo es conseguir la bidireccionalidad sin los inconvenientes que HTTP presentaría para obtenerla, como son el mayor número de conexiones por cliente desde el agente de usuario (no sólo una), o como el *overhead* que añade, y con el beneficio de poder utilizar la infraestructura existente (*proxies*, filtrado, autenticación...). Por tanto el protocolo es, conceptualmente, una capa sobre TCP independiente de HTTP, no basado en él, exceptuando que el *handshake* se realiza mediante el mecanismo HTTP *Upgrade*, y que tiene alguna característica en común como emplear los puertos 80 y 443, este último para comunicaciones cifradas.

Una vez establecida la conexión, los mensajes se intercambian en paquetes, *frames*, con una pequeña cabecera y la carga puede ser binaria o texto codificado en UTF-8 [63, Section 5.6].

El API de *Websocket* es un sencillo conjunto de métodos y variables de es-

tado que permiten manejar un *WebSocket* desde un agente de usuario. En concreto ofrece:

- un constructor `Websocket()` y un método de finalización `close()`. Al constructor hay que especificarle como parámetro la URL del servidor, similar a las de HTTP, en las que cambia el protocolo del esquema por `ws` o `wss` (socket seguro). Una vez conectado, podemos consultar el estado de la conexión en la propiedad `readyState`, que será uno de cuatro posibles: `Connecting`, `Connected`, `Closing` o `Closed`.

- métodos para el manejo de eventos: `onOpen()`, `onError()`, `onMessage()` y `onClose()`. Cuando se reciben datos, el atributo `binaryType` es el que indica cómo deben ser tratados esos datos, por ejemplo, como `blob` o como un `arraybuffer`.

- la manera de enviar datos con `send()` y saber con el atributo `bufferedAmount` cuántos están encolados a la espera de enviarse.

Adobe® Flash® Socket

La empresa Adobe®, con su tecnología Flash®, provee un entorno multiplataforma para la creación y ejecución de aplicaciones y contenido multimedia para sistemas operativos y, sobre todo, navegadores web. Por tanto, las páginas web pueden contener embebido contenido Flash, archivos de extensión `.swf`, con el que el usuario puede interactuar a través del *browser*.

Este contenido puede programarse con un lenguaje de programación, ActionScript, que, entre una variada gama de objetos, ofrece su propia implementación de sockets TCP. A través ésta, Socket.IO ofrece, para la comunicación con el servidor, su propio archivo `.sfw` que simula un *WebSocket*. Este archivo está incluido en el módulo Socket.IO y en caso de ser necesario, se sirve automáticamente al cliente que lo insertará dinámicamente en la página web.

ActionScript y la implementación que se hace de WebSocket en este lenguaje queda fuera del campo de estudio del proyecto y no se profundizará más en él. Es importante reseñar que los sistemas operativos móviles de Apple no permiten este tipo de contenido y, desde Agosto de 2012, la tienda de aplicaciones Google Play no dispone del intérprete de Flash para Android, con lo que su uso está muy limitado.

AJAX Long Polling

Una técnica usada para simular una interacción con el servidor en tiempo real es *AJAX Polling*, consistente en abrir una conexión *AJAX* cada cierto intervalo de tiempo con objeto de recibir datos del servidor. Si el intervalo es lo suficientemente pequeño, la sensación es la de comunicación en tiempo real. Sin embargo, el coste que introduce reducir este intervalo es:

- Latencia HTTP: establecer una conexión remota conlleva un tiempo de latencia del orden típicamente de los 200 milisegundos pudiendo ser mucho mayor según qué circunstancias. En este escenario se obtiene una degradación del rendimiento de la aplicación, lo cual se agrava si el servidor no genera nuevos datos.

- Desperdicio de recursos: si el servidor no tiene datos que enviarnos, se produce tráfico de red y ciclos de petición/respuesta para nada.

Para optimizar ambas desventajas, se introduce una modificación en la manera que se realizan las peticiones *AJAX*. En este caso, si el servidor no tiene nada que enviar, la conexión se mantiene abierta esperando, como en *AJAX Polling*. En el momento que envía algo y se procesa, en lugar de esperar un intervalo de tiempo para establecer la conexión, se realiza inmediatamente otra petición *AJAX*. De esta manera se reduce drásticamente la latencia y el tráfico de red anteriormente mencionados. Esto es lo que se conoce como *AJAX Long Polling* [65].

AJAX Multipart Streaming

Esta técnica permite mantener una conexión abierta con el servidor si se fija el tipo MIME de la respuesta como `multipart/x-mixed-replace`. Este tipo lo introdujo Netscape en 1995 como una característica de su navegador a la que llamo *server push* con la que pretendía que habilitar a los servidores a enviar nuevas versiones de los elementos de una página web.

Sin embargo, los únicos navegadores que aceptan esta técnica son los que están basados en Gecko, el motor de Mozilla, como, por ejemplo, Firefox [66].

Forever Iframe

Es otra de las técnicas que permiten mantener abierta una conexión permanente con el servidor empleando una etiqueta html `<iframe>` de modo que se permite embeber una página web dentro de otra. Esta página se envía del servidor con dos cabeceras concretas:

Con esto se consigue que el navegador considere la página como un ente infinito (de ahí el nombre *Forever Iframe*) y, como todos los navegadores modernos, la interpretará según la vaya recibiendo. Conociendo esto, conforme haya datos disponibles, el servidor enviará etiquetas `<script>` en las que se llame a una función del frame padre que será la que reciba y procese la información del servidor.

Esta técnica crea un canal unidireccional Servidor-Cliente. La comunicación Cliente-Servidor se realiza a través de peticiones HTTP POST normales sobre conexiones *Keep-Alive* [67].

Un punto negativo de este método es que altera la experiencia de usuario porque la existencia de este *iframe* supone que el navegador muestre continuamente un icono de carga o la barra de progreso.

JSONP Polling

Esta técnica de *Polling* utiliza JSONP en lugar de JSON. JSONP significa *JSON with Padding* [68]. Se emplea para evitar la limitación impuesta por los navegadores con su política de "mismo dominio de origen" por la cual el navegador restringe el intercambio de datos entre una página web y un servidor sólo si ambos pertenecen al mismo dominio, con objeto de evitar problemas con código Javascript.

Se solicita al servidor que "recubra" los datos con una llamada a una función presente en el espacio de ejecución del navegador capaz de procesarlos correctamente como la siguiente:

```
function handleData (datosJSON){
  // Procesamiento de los datos del servidor
}
```

Este recubrimiento es a lo que se llama *padding* y se incluye como parámetro en la *query* de la URL que proporciona los datos, por ejemplo:

```
http://dataserver.com/webservice?callback=handleData
```

Esta URL se emplea como fuente, atributo `src`, en una etiqueta `script` de HTML [69] porque esta etiqueta no posee la restricción que se impone al resto de peticiones `cross-domain`.

El resultado de la petición para este ejemplo concreto debería ser:

`handleData( {datosJSON} )`

que se inserta y ejecuta dentro de la etiqueta `<script>` invocando al *callback* que se ha establecido en la página. Este *callback* procesa el resultado evitando los errores que se producirían si el servidor simplemente hubiera devuelto el objeto `datosJSON`, porque hubiera sido interpretado como un bloque de código (está entre llaves), no como un objeto, por otra parte, innacesible al resto del código de la página por no estar asignado a variable alguna [70].

De esta manera, aunque no es una técnica que elimine el riesgo de código malicioso, puesto que puede existir un servicio web comprometido, se resuelve el problema de estandarización *cross-domain* [71].

7.2. Usando Socket.IO

7.2.1. Servidor

La manera de utilizar Socket.IO en la parte de servidor con Node es importándolo con `require`, como con todos los módulos, y asociándolo a un servidor HTTP, que es uno de los parámetro que acepta el método `listen()` (útil sobre todo si se trabaja con otros módulos para desarrollo web como Express):

```
var server = require('http').createServer(),
    io = require('socket.io').listen(server);
server.listen(8080);
```

O bien indicando al método `listen()` el puerto donde escuchar y él mismo se encarga de crear el servidor:

```
var io = require('socket.io').listen(8080);
```

A partir de entonces la aplicación está preparada para recibir conexiones y hacer disponible el conjunto de los sockets conectados en la propiedad `io.sockets`. Todos estos *sockets* poseen varios atributos, sobre todo referentes al entorno y al estado de la conexión, de los cuales destaca uno, el identificador de cliente *id*, accesible como: `socket.id`. Es único y se genera en el proceso de *handshake* con el cliente, por lo que es conocido por ambos.

Antes de ello, el cliente debe superar un proceso de autenticación, en la primera fase del *handshake*. Este proceso es opcional y configurable por el programador asignando una función al método `authorization()`:

```
io.configure(function(){
  io.set('authorization', function(handshakeData, callback){
    // Aqui el algoritmo de autenticacion
    callback(null, true);
  });
});
```

La función recibe todos los datos referentes al *handshake* en `handshakeData`, que es un objeto que la librería genera en este proceso, en los cuales puede basarse para autorizar o no la conexión. Una vez determinado esto, se invoca al *callback* cuyo primer argumento es un error, en caso de haberse producido, y el segundo indica si el proceso ha sido satisfactorio, `true`, o no, `false`. En el primer caso, el cliente recibirá un evento `'normal connect'` y en el último caso el evento que se dispara es `'error'`.

Indicar que, durante la autenticación, se puede manipular el objeto `handshakeData` añadiendo o quitando atributos. Estas modificaciones quedarán reflejadas posteriormente puesto que este objeto se sigue usando después del proceso. Los atributos que se añadan estarán luego disponibles en el propio *socket*, en `socket.handshake`, durante el evento `'connecting'`. Los que se quiten, y es importante remarcarlo, dejarán de estar disponibles para futuras acciones sobre ellos, como por ejemplo procesos de autenticación posteriores [72].

La librería permite atender a eventos generados en los *sockets* autorizados. Estos eventos se pueden separar en dos grupos: los definidos por Socket.IO y los que define el usuario. Los tres primeros son [73]:

`io.sockets.on('connection', function(socket){})`
: Es la manera que existe de obtener la referencia a un *socket* la primera vez que se conecta. Una vez se tenga, a través del *callback*, se pueden establecer en él los eventos que se deseen.

`socket.on('message', function(message){})`
: Se emite cuando se recibe un mensaje a través del método del *socket* cliente `socket.send()`. El *listener* del evento recibe como argumento el mensaje que envía el servidor.

`socket.on('disconnect', function(){})`
: Se espera que se emita este evento cuando un *socket* se desconecta, en principio porque el cliente desconecta aunque también es posible que se emita cuando lo haga el servidor.

La manera que tiene el programador de definir eventos es sencilla:

`socket.on('evento', [function(datos, callback){}])`
: `evento` se sustituirá por el nombre que se le haya dado. Cada vez que el cliente emita uno de ellos se ejecutará la función anónima.

  Esta función recibe como primer argumento los datos que envía el cliente, si acaso envía, para procesarlos.

  El segundo argumento es un *callback* opcional. De existir, es una función a la que se llama pasándole los argumentos que admita con los valores que se deseen. Su misión es la de asentimiento: se invoca en el servidor y se ejecuta en el cliente, que es donde está definida y donde, por tanto, produce efecto.

Además, en los *sockets* se puede almacenar información asociada al cliente que se puede gestionar con dos métodos accesores:

`socket.set('propiedad', propertyValue , function(){})`
: Asigna la `propiedad` al `socket` y le asigna el valor `propertyValue`. El *callback* se ejecuta una vez se realiza el proceso de asignación.

`socket.get(`'propiedad', function(error, propertyValue){})`
: Intenta obtener del `socket` el valor de la `propiedad` ejecutando el *callback* tanto si lo consigue, pasándoselo en el parámetro `propertyValue`, como si no, especificando el error en el parámetro `error`.

Estos métodos, y, por tanto, la información almacenada a través de ellos, sólo son accesibles desde la parte del servidor, no desde el cliente.

En muchas ocasiones, será necesario o más útil y/o cómodo trabajar con colecciones de *sockets* agrupados según algún criterio específico. Se ofrece para ello dos maneras de particionar el conjunto de total: mediante *namespaces* o mediante *rooms* (o salas).

Namespaces
: se emplean cuando se quiere que los mensajes y eventos se emitan en un *endpoint* en particular que se especifica como parte de la URI empleada por

el cliente cuando se conecta, por ejemplo, `http://example.com/chat`. El beneficio que se obtiene es el de multiplexar una única conexión, es decir, en lugar de usar varias conexiones *Websocket*, se usará la misma.

El cliente solicita la conexión al *endpoint* indicándolo en el método `connect()`. El servidor manejará los diferentes *endpoints* con el método `io.of('endpoint')` que devuelve el *endpoint* indicado. Sobre éste se podrá trabajar como si del conjunto global de *sockets* se tratara. Sirva de ejemplo:

```
varnamespace1 = io
  .of('/endpoint_1')
  .on('connection', function(socket){
    namespace1.emit('event', { newSocket: socket.id });
  });

var namespace2 = io
  .of('/endpoint_2')
  .on('connection', function(socket){
    socket.emit('event', { hello: socket.id });
  });
```

Al igual que en el espacio global, sobre un *namespace* se puede definir un proceso de autenticación. Se hace a través del método encadenable `authorization()` del *namespace*:

```
var namespace = io
  .of('/endpoint')
  .authorization(function(handshakeData, callback){
      handshakeData.miToken = 'autorizado';
      callback(null, true);
  })
  .on('connection', function(socket){
      console.log(socket.handshake.miToken);
  });
```

En caso de que falle la autenticación, a diferencia de las *rooms*, los *namespaces* no emiten el evento `'connect_failed'` que se puede manejar en el cliente. No varía en caso de éxito donde el cliente seguirá recibiendo un

evento `'connect'`.

Rooms

[74] básicamente son etiquetas que se asignan a los *sockets* para crear grupos de ellos pudiendo un *socket* tener varias. Es una manera fácil de realizar el particionado de los clientes. A diferencia de los *namespaces*, en los que el cliente solicitaba formar parte, esta división se realiza en el servidor como mecanismo disponible para el programador y transparente al cliente .

Otras diferencias con los *namespaces* son que no tiene soporte nativo para la autenticación, y que las salas son parte de un *namespace* mientras que los *namespaces*, sin embargo, son parte del *scope global* [75].

El manejo de las salas es extremadamente sencillo:

- el método `socket.join('room')` añade un cliente que se eliminará con `socket.leave('room')`
- el método `io.sockets.in('room').emit('event', data)` genera un evento que reciben todos los clientes de esa sala

Desde el código del servidor, se tiene disponibles la lista de *rooms* que se van creando en el atributo `io.sockets.manager.rooms`. Esta tabla *hash* contiene las salas como claves y, asociado a cada una de estas claves, el array de clientes pertenecientes a esa *room*. No obstante resulta más sencillo acceder a salas y clientes con los métodos:

`io.sockets.clients('room')`
: para obtener las conexiones a una *room*

`io.sockets.manager.roomClients[socket.id]`
: para obtener la lista de salas a las que pertenece un cliente determinado. La lista es un objeto cuyos atributos, todos de valor `true`, son el nombre de cada una de las salas.

Hay que destacar que el nombre de las salas manejado de las maneras anteriores aparecerá precedido del carácter `/`.

7.2.2. Cliente [2]

Del lado del cliente, Socket.IO se importa como cualquier fichero `.js`:

```
<script src="/socket.io/socket.io.js"></script>
```

En este caso, será la propia Socket.IO la que se encargue de servir la librería al cliente siempre que la aplicación web y Socket.IO corran en el mismo servidor y puerto. Para evitar problemas de caché con el navegador, esta forma admite número de versión de esta manera:

```
<script src="/socket.io/socket.io.v0.9.10.js"></script>
```

En caso contrario:

```
<script src="http://<uri:port>/socket.io/socket.io.js"></script>
```

Yendo más allá, se puede especificar otra ubicación para la librería y servirla por cualquier otro medio, teniendo en cuenta que hay que:

1. clonar el proyecto correspondiente al cliente, *socket.io-client*, desde github[1]: `git clone git://github.com/LearnBoost/socket.io-client.github`
2. copiar el contenido de `/dist` en la ubicación desde donde se va a a servir. Para la versión 0.9.10 son cuatro ficheros. A saber: `WebSocketMain.swf`, `WebSocketMainInsecure.swf`, `socket.io.js` y `socket.io.min.js`

Una vez inicializada la librería, se puede trabajar con un *socket* conectándolo al servidor:

```
var socket = io.connect(host, options)
```

El `host` es, lógicamente, el servidor donde corre la aplicación basada en Socket.IO, aunque la función `connect()` está programada de tal manera que si no se proporcionan argumentos, se realizará un autodescubrimiento del servidor basándose en la URL del documento que se ha obtenido del servidor.

El argumento `host` también puede indicar a qué *namespace* va a conectarse el *socket* simplemente indicando el *endpoint* como parte de la URI o incluso directamente, ya que `connect()` autodetectará el *host*.

```
var chat = io.connect('http://example.com/chat');
var news = io.connect('/news');
```

Conectarse a un *namespace* limitará los mensajes y eventos que se reciban a los emitidos por los clientes en dicho *endpoint*.

[1] https://github.com/

A través del segundo argumento (opcional), `options`, se pueden modificar un amplio rango de parámetros relativos a la conexión. Por defecto, el objeto que contiene las opciones es:

```
{
  "force new connection": false,
  "resource": "socket.io",
  "transports": ["websocket",
                 "flashsocket",
                 "htmlfile",
                 "xhr-multipart",
                 "xhr-polling",
                 "jsonp-polling"],
  "connect timeout": 5000,
  "try multiple transports": true,
  "reconnect": true,
  "reconnection delay": 500,
  "reconnection limit": Infinity,
  "max reconnections attempts": 10,
  "sync disconnect on unload": false,
  "auto connect": false,
  "flash policy port": 10843,
  "manualFlush": false
}
```

El significado de los distintos parámetros y su utilidad se detallan a continuación:

`force new connection`
: cuando se añade como opción, permite a un cliente crear varias conexiones bajo distintos *sockets*, de otra manera, si se intentase abrir una conexión se trabajaría con la ya establecida.

`resource`
: identifica el *endpoint* donde realizar conexiones, por defecto es el espacio global `socket.io` pero puede ser el de cualquier *namespace*.

`transports`
: es la lista por orden de preferencia de transportes que se quiere que se

intente utilizar para realizar la conexión. Por defecto, están todos, pero puede restringirse a unos cuantos.

`connect timeout`
: son los milisegundos que tiene Socket.IO para conectarse al servidor mediante un transporte antes de reintentarlo. Hay que tener en cuenta que algunos transportes necesitan más tiempo que otros para conectar por lo que conviene no modificar este parámetro.

`try multiple transports`
: *flag* que autoriza los intentos de conexión con el resto de modos de transporte en caso de `'connect timeout'`.

`reconnect`
: *flag* que indica si se debe o no intentar la reconexión automática en caso de desconexión.

`reconnection delay`
: factor de espera en milisegundos antes de volver a intentar una reconexión con el servidor. Forma parte de un algoritmo de tipo exponential *back off* que fija los milisegundos totales del siguiente intento según la fórmula $N \times delay$ con N=1, 2, 3... número de intento.

`reconnection limit`
: como el anterior parámetro, `reconnection delay` crece según un algoritmo exponencial a cada intento de reconexión, se introduce éste otro parámetro, `reconnection limit`, para marcar un límite máximo, también en milisegundos, que marca hasta dónde puede aumentar.

`max reconnections attempts`
: número de intentos de reconexión a través del modo de transporte elegido antes de realizar el definitivo y, en caso de volver a fallar, empezar a probar con el resto de modos de transporte seleccionados.

`sync disconnect on unload`
: realiza una desconexión del *socket* antes de detener la ejecución de la librería, como sucede, por ejemplo, al cerrar o recargar la página web en el navegador. Por defecto es false, con lo que el servidor detectará a través de los *heartbeats* que se ha interrumpido la conexión.

`auto connect`

conecta automáticamente con el servidor cuando se crea el *socket*.

`flash policy port`
: configura el puerto donde se consulta el archivo de políticas de Flash (*flash policy file*) [76]. Estos archivos definen qué privilegios de conexión alcanzan los *sockets* que crea Flash, por tanto sólo aplica en caso de que el transporte sea `flashsocket`. Flash usa por defecto el puerto 843, pero Socket.IO lo ha cambiado a 10843 para que no sea un puerto de *root* (aquellos que están por debajo de 1024).

`manualFlush`
: activando esta opción se deja a criterio del programador cuándo invocar el método `socket.flushBuffer()` que envía los datos al servidor.

Cuando se invoca el método `connect()` comienza el proceso de *handshake*, definido en el protocolo de Socket.IO [77], entre cliente y servidor que se inicia con una petición POST automática a éste del tipo

```
http://example.com/socket.io/1/
```

El servidor reconocerá el intento de establecer una conexión por la ausencia de los campos `transport id` y `session id` propios de una URL perteneciente a una conexión ya establecida.

Si la negociación ha sido satisfactoria se recibirá un código `200 OK` con un cuerpo de respuesta que incluye, separados por dos puntos `':'`, cuatro campos:

- El identificador de sesión, `session id`, que es el mismo que identifica al *socket* en el servidor, `socket.id`
- Opcionalmente, el *timeout* en segundos de espera de los *heartbeats*, `heartbeatTimeout`. Su ausencia en la respuesta indica que ni cliente ni servidor esperan *heartbeats*
- El *timeout* en segundos de cierre de la conexión `connection closing timeout`. Marca cuanto tiempo tarda en cerrarse el transporte una vez que el *socket* se considera desconectado
- La lista de transportes soportados por el servidor

Por ejemplo, `4d4f185e96a7b:15:10:websocket,xhr-polling`.

Una vez recibido la respuesta a la negociación el cliente selecciona el modo de transporte intersecando la lista de los que puede usar él y la lista que recibe del servidor e intentando conectar con ellos uno a uno hasta tener éxito. Normalmente debería ser el primero, pero puede depender, por ejemplo, de si la conexión es *cross-domain*.

Completado este proceso, la forma de las URL que utiliza Socket.IO es como sigue:

```
 <esquema>'://' <host>'/' <namespace>'/' <versión protocolo>'/' <id transporte>'/' <id sesión>'/' ['?'<query>]
```

esquema
: indica el protocolo que se usa, en principio, `http` o `https`, pero puede actualizarse a `ws` o `wss`

host
: la máquina que ejecuta el servidor Socket.IO, por defecto será aquella que haya servido la página cargada por el cliente

namespace
: el *namespace* donde se conecta el cliente. Si éste no indica ninguno, por defecto será `socket.io`

versión protocolo
: hasta el momento, el protocolo Socket.IO está en la versión 1. No confundir con la versión del módulo, que la que se emplea aquí es la 0.9.10

id transporte
: el identificador del transporte que se haya acordado sobre el que realizar la conexión: `xhr-polling`, `xhr-multipart`, `htmlfile`, `websocket`, `flashsocket`, `jsonp-polling`. Cada uno, como es obvio, se corresponde con uno de los métodos detallados al inicio.

id sesion
: es el identificador de cliente que el servidor asigna al *socket* durante el proceso de *handshake*.

query
: es un parámetro opcional a través del cual podemos pasar información adicional al servidor respetando las reservadas por Socket.IO:

t

timestamp, usado para evitar la caché en viejos agentes de usuarios

disconnect

usado para la desconexión

En el cliente son conocidas las claves que se envían en la *query* y, para acceder a ellas desde el servidor, se pueden consultar como atributos del objeto `socket.handshake.query`.

Pueden sin embargo recibirse otros dos códigos que indican que el proceso no se ha terminado de realizar satisfactoriamente, son:

401 Unauthorized

Cuando el servidor rechaza la conexión en base a las credenciales (cabecera `Cookie` o cualquier otro método) que presenta el cliente para ser autorizado.

503 Service Unavailable

Cuando se rechaza la conexión por razones como exceso de carga en el servicio.

Una vez se tenga una instancia de *socket* conectada se tendrá información de su estado a través de varias propiedades:

options

el objeto que contiene todas las opciones fijadas para el *socket*

connected

booleano que indica si el *socket* está conectado o no

connecting

booleano que indica si el *socket* está conectando o no

reconnecting

booleano que indica si el *socket* está reconectando o no

transport

referencia a la instancia del transporte que se emplea en la comunicación

y se puede utilizar a través de los métodos:

send(mensaje, function(){})

Envía la cadena de datos `mensaje`. Cuando el servidor la recibe, se ejecuta

automáticamente la función de asentimiento pasada como segundo argumento.

`emit('evento', {datosJSON}, function(params){})`
Genera el `'evento'` en el *endpoint* donde está conectado y envía al resto de clientes en el *endpoint* los datos en notación JSON `datosJSON`. El tercer argumento es una función de asentimiento que puede invocarse desde el código del servidor, por decisión del programador, a diferencia del asentimiento de `send()` que, como se ha comentado, es de ejecución automática.

`disconnect()`
Finaliza la conexión con el servidor.

De la misma manera que en el servidor, los *sockets* en la parte cliente pueden reaccionar a múltiples eventos, algunos definidos por Socket.IO y otros por el programador de la aplicación. Entre los ya establecidos[2]:

`socket.on('connect', function(){})`
Emitido cuando el `socket` se conecta satisfactoriamente.

`socket.on('connecting', function(transport_type){})`
Cuando el `socket` está intentando conectar con el servidor, tanto la primera vez como en una reconexión. Se facilita por parámetro el tipo de transporte a través del cual se está intentando realizar la conexión. Si el `socket` está intentando reconectarse, este evento se emite después del evento `'reconnecting'` y antes del evento `'connect'`.

`socket.on('connect_failed', function(){})`
Se emite cuando expira el *timeout* después del último intento de conexión si está activada la opción `connect timeout`. Si `try multiple transports` es la opción activada sólo se emite una vez probados todos los posibles medios de transporte.

También se emite cuando el mecanismo de autenticación, definido por el programador, de un *namespace* deniega el acceso a dicho *namespace*.

`socket.on('message', function(message){})`
Emitido cuando se recibe un mensaje del servidor.

`socket.on('close', function(){})`
Se emite cuando se cierra la conexión. Los autores de la librería nos advier-

ten de que puede emitirse bajo desconexiones temporales conocidas, como sucede con las técnicas de *Polling*, donde se realiza una desconexión cuando se han recibido los datos del servidor para inmediatamente abrir otra que espere los siguientes.

`socket.on('disconnect', function(){})`

Cuando el socket se desconecta.

`socket.on('reconnect',function( transport_type, reconnectionAttempts){})`

Se emite una vez la conexión se haya reestablecido y sólo si está activada la opción `reconnect`. Los argumentos que recibe el listener son `transport_type`, que indica con qué modo de transporte se ha realizado la reconexión, y `reconnectionAttempts`, que indica el número de intentos realizados para reconectar.

`socket.on('reconnecting', function(reconnectionDelay, reconnectionAttempts){})`

Se emite en cada uno de los intentos de reconexión, e indica el número de intentos que lleva y el tiempo de espera hasta el siguiente.

`socket.on('reconnect_failed', function(){})`

Emitido cuando han fallado todos los intentos de reconexión y no ha podido establecerse la reconexión con el servidor.

Pero no es `on()` el único método para trabajar con eventos. *Socket* tiene disponible también:

`once('evento', function(){})`

Permite atender a un evento una sola vez después de lo cual se elimina el *listener*.

`removeListener('evento', functionName)`

Elimina el manejador de eventos `functionName` del `'evento'`. `functionName` no puede ser, por tanto, una función anónima.

7.3. Aplicación con Socket.IO

Para comprender mejor el funcionamiento de la librería se va a implementar una versión de un juego sencillo de memorización.

7.3.1. Descripción del juego

El juego es un clásico de los juegos de mesa. Consiste en una serie de cartas con una imagen, iguales dos a dos, que se colocan boca abajo sobre un tablero. Por turnos, los participantes deben descubrir dos de ellas buscando que sean pareja. Si las cartas descubiertas en el turno no son iguales, se vuelven a dar la vuelta y se cede el turno al otro participante. Si son iguales, se dejan a la vista y el jugador vuelve a descubrir otras dos. Los jugadores deben intentar memorizar la posición de las cartas que se van descubriendo con objeto de conseguir descubrir el mayor número de parejas posibles siendo el ganador el que más parejas averigüe.

7.3.2. Objetivos perseguidos

Con la realización de la aplicación se persigue conocer los siguientes aspectos del módulo Socket.IO:

- creación de *sockets* y subscripción de los mismos a eventos, tanto del sistema como definidos por las necesidades de la aplicación
- funciones de asentimiento en cliente
- almacenamiento de datos asociados con el cliente durante la comunicación
- utilización de *rooms* para agrupar y gestionar clientes de una manera sencilla, donde la gestión implica:
    - agregar un cliente a una sala
    - emitir eventos en la sala

7.3.3. Diseño propuesto

De acuerdo con las sencillas reglas explicadas anteriormente, una partida del juego puede modelarse como un objeto *Game* que contiene el estado del juego entre los dos participantes. Consecuentemente la información que una instancia este objeto almacena es:

- qué cliente jugó el último turno, `lastTurn`
- cuál es la última carta que se ha descubierto en ese turno, `lastCard`
- el orden de las cartas del tablero, `cardsMap`. Este orden se determina al azar: en el constructor de la clase, se inspecciona el directorio de imágenes y a cada una de ellas se le asignan dos identificadores unívocos

Un nuevo juego comienza cuando se conecta un jugador mediante un navegador web. A través de un formulario de la página, envía su nombre de usuario cuando pulsa el botón 'Play!' lo que origina una nueva conexión en el servidor.

Ver "Implementación con Socket.IO. Punto 1"

Este cliente genera un evento joins, que señala que quiere unirse a una partida con su nombre de usuario:

```
joins = { username }
```

Ver "Implementación con Socket.IO. Punto 2"

Si el nombre no se acepta, por ejemplo porque el otro participante se llame igual, porque el cliente intente conectarse dos veces con el mismo nombre o porque se haya recibido un nombre en blanco, se produce un error en forma de evento que supone fin de conexión.

```
error = { code, extendedInfo }
```

Ver "Implementación con Socket.IO. Punto 3"

Si el nombre es correcto, se inicializan los datos del cliente, concretamente nombre y puntuación, guardándolos en el *socket*

Ver "Implementación con Socket.IO. Punto 4"

y se prepara la partida comprobando la condición de inicio de la misma y se asiente enviando al cliente su propio nombre como confirmación.

Ver "Implementación con Socket.IO. Punto 5"

La consideración que se debe tener en cuenta para comenzar una partida es si hay o no otro jugador esperando.

- No hay nadie esperando: el sistema tendrá conocimiento de que no hay un cliente a la espera porque la variable `waitingRoom` es `null`. Se genera entonces una nueva partida instanciando el objeto *Game*. Esta instancia se almacena en un *hashmap* (un simple objeto Javascript) asignándole una clave o identificador de juego unívoco, obtenido gracias al módulo *node-uuid*, y se actualiza `waitingRoom` con esa clave.

  ```
  if (!waitingRoom){
    startingPlayer = username;
    waitingRoom = uuid.v1();
    room2game[waitingRoom] = new models.Game();
    // TODO: Agregar el socket a una sala
  }
  ```

  Además, se utilizará el identificador, que servirá de etiqueta, para crear una sala donde agrupar a los dos clientes para que se desarrolle el juego.

  Ver "Implementación con Socket.IO. Punto 6"

- Si hay alguien esperando: el sistema lo sabrá porque `waitingRoom` contiene un identificador de juego a través del cual podemos añadir al nuevo cliente a la sala creada para la partida e, inmediatamente, comenzar el juego.

  ```
  room2game[waitingRoom].lastTurn = username;
  // TODO: Agregar el cliente a la sala
  // TODO: Generar el evento de inicio 'start'
  waitingRoom = null;
  startingPlayer = null;
  ```

Se marca el inicio a través del evento `'start'`, difundido a ambos participantes. Con él se informa de los nombres de los éstos en orden de conexión ya que el que lo haya hecho primero será quien comience descubriendo las cartas.

```
start = { [players] }
```

Ver "Implementación con Socket.IO. Punto 7"

El desarrollo del juego transcurre según las normas anteriormente descritas: por turnos se van descubriendo cartas de dos en dos intentando formar parejas para puntuar. Cada pareja que se descubra proporciona dos puntos al jugador y, además, la oportunidad de descubrir otro par de cartas.

Las interacciones entre los usuarios y el servidor de juegos se producen como intercambio de mensajes. Se modelarán estos mensajes como eventos que se disparan en el *socket*, bien por cada uno de los clientes o bien por el servidor en respuesta a ellos. De ellos, el principal es el evento `'discover'`, que tanto el cliente como el servidor pueden emitir.

Emitido por el cliente, el evento discover envía al servidor el identificador de carta elegida por el jugador y que es la posición de la misma en el tablero.

```
discover = { id }
```

El código de manejo del evento en el servidor contiene la lógica del desarrollo del juego y gestiona discover recuperando, gracias al identificador del *socket*, en qué *room* se ha generado

Ver "Implementación con Socket.IO. Punto 8"

A través de ésta, se conocerá cual es el estado de la partida: si no ha acabado o si la carta no está descubierta ya, en cuyo caso se habrá borrado del atributo del tablero `cardsMap` de la instancia de *Game*, que traduce el identificador a una de las imágenes.

```
var id = card.id
var game = room2game[ roomId ];
if (game == undefined || !(id in game.cardsMap)) return;
```

Sólo se procesarán los eventos `'discover'` que provienen del jugador que tiene el turno, ignorando el resto. Sabremos qué usuario es gracias al *socket*

Ver "Implementación con Socket.IO. Punto 9"

Ver "Implementación con Socket.IO. Punto 10"

El servidor emite el evento `'discover'` a los jugadores cuando ha verificado lo anterior y ha realizado la traducción del identificador. La información que contiene el evento es la ruta de la imagen para que los clientes puedan mostrarla en los navegadores.

```
discover = { id, src }
```

Ver "Implementación con Socket.IO. Punto 11"

Si se ha descubierto la primera carta del turno, el modelo de juego memoriza cuál ha sido, no por su identificador sino por la ruta de la imagen en el servidor, y queda a la espera de la siguiente.

```
var lastId = game.lastCard;
if (lastId == null || lastId == id) {
  game.lastCard = id;
  return;
};
```

Si es la segunda carta la que se descubre, una vez traducida, se compara con la recién descubierta. Aquí sólo caben dos opciones:

- que las cartas sean diferentes: se pasa turno, actualizando el modelo de juego con el cliente que ha jugado el turno, `lastTurn`, y se limpia `lastCard`, primera carta que salió en la mano.

  ```
  game.lastTurn = username;
  game.lastCard = null;
  ```

  Hecho esto se envía el mensaje de fallo, fail, que

  ```
  fail = { [ids], src }
  ```

  Ver "Implementación con Socket.IO. Punto 12"

- que sean iguales: se eliminan del `cadsMap` del juego los `id` de las cartas, por si por algún error se envía alguna al servidor, poder ignorar la petición comprobando su presencia:

  ```
  if (game.cardsMap[lastId] == game.cardsMap[id]){
    delete game.cardsMap[lastId];
    delete game.cardsMap[id];
    game.lastCard = null;
    // TODO: Generar evento de acierto
    // TODO: Recalcular puntuacion y notificarla a los clientes
  }
  ```

Se notifica a los clientes mediante el evento `'success'`, que incluye las `id` de la pareja. El cliente debe dejarlas descubiertas en la interfaz.

```
success = { [ids] }
```

Ver "Implementación con Socket.IO. Punto 13"

Posteriormente, se actualiza la puntuación del jugador sumando dos puntos y se notifica a los jugadores con el evento `'score'`, que informa de la puntuación actualizada del jugador que ha ganado el turno:

```
score = { username, points }
```

Ver "Implementación con Socket.IO. Punto 14"

Después de cada turno, se comprueba si el juego ha terminado consultando el modelo. La condición para que haya acabado es que se hayan eliminado todos los identificadores del `cardsMap`, comprobación que se deja a cargo del objeto *Game*.

```
if (game.isOver()) {
  // TODO: Notify the end to clients
  delete room2game[roomId];
}
```

Si se ha finalizado, se comunica a los jugadores mediante `'finish'`, que es un mensaje sin contenido.

```
finish = {}
```

Ver "Implementación con Socket.IO. Punto 15"

7.3.4. Implementación con Socket.IO

Los puntos concretos de la aplicación que se refieren a Socket.IO se describen en detalle a continuación:

1. El servidor recibe una nueva conexión
   Cuando se recibe una nueva conexión, Socket.IO genera uno de los pocos eventos definidos en el servidor al que el código deberá atender:

```
io.sockets.on('connection', function(socket){
  socket.on(...);
  socket.on(...);
});
```

Este evento proporciona el *socket* de la conexión que se podrá manipular, básicamente para añadir los manejadores al resto de eventos que la parte cliente pueda generar en él.

2. Evento de usuario `'joins'`
   Este evento está definido por el programador. Notifica cuándo un cliente solicita unirse a una partida. Tiene como mensaje asociado el nombre de usuario del jugador.

```
socket.on('joins', function(message, callback){
  var username = message.username;
  // TODO: Gestionar la incorporacion a un nuevo juego
});
```

3. Errores
   A lo largo de la aplicación, se pueden encontrar situaciones que no puedan manejarse con el código y sea necesario notificar al cliente mediante el evento `'error'`. Se han definido uno para el juego que se genera mediante `emit()` y se produce cuando cliente no proporciona nombre de usuario o se conectan dos clientes con el mismo nombre:

```
socket.emit('error',
  { code: 0, extendedInfo: 'No username provided' }
);
```

4. Almacenando datos asociados a un cliente en el *socket*
   En los *sockets* de los clientes se puede almacenar información como si de una sesión HTTP se tratara. Para el juego la utilidad es poder guardar el nombre de usuario y su puntuación actualizada.

```
socket.set('username', username);
socket.set('score', 0);
```

5. *Callback* de asentimiento
   El segundo argumento de la función que maneja el evento `'joins'` es un *callback* de asentimiento. Cuando el cliente genera un evento con

`emit('joins', message, callback)` da opción al servidor a invocar una función *callback* que se ejecutará en el propio cliente.

En el caso del juego, se emplea para asentir que el nombre de usuario se ha recibido y aceptado sin errores:

```
socket.on('joins', function(message, callback){
  var username = message.username;
  if (username == '' || username == null){
    // TODO: Generar evento de error
  } else {
   // TODO: Chequear condiciones de inicio de partida
   // Asentimos al cliente
   callback(username);
  }
});
```

6. Creación de *rooms* y asignación de *sockets*
   Las salas se crean dinámicamente cuando se agrega el primer *socket* a ella. Por tanto bastará con:

   ```
   socket.join(waitingRoom);
   ```

7. Evento de usuario `'start'`
   `'start'` es un evento definido por el programador para avisar a los clientes de que comienza el juego. Una vez conoce el nombre de ambos participantes, el servidor lo emite a los dos integrantes de la sala invocando a `io.sockets.in()`. Envía los dos nombres como parámetro, siendo el primero de ellos el que comienza el juego:

   ```
   io.sockets.in(waitingRoom).emit('start',
     { players: [startingPlayer, username]}
   );
   ```

8. Gestión de *rooms* asociadas a *sockets*
   Cuando se necesita obtener información acerca de las salas a las que pertenece un determinado cliente, se debe hacer uso de ciertas estructuras del *Manager* de Socket.IO. En concreto del *hashmap* `io.sockets.manager.roomClients` que contiene, asociado a cada identificador de cliente, arrays de etiquetas de todas las salas a las que pertenece

el citado cliente.

Señalar que a cada una de esas etiquetas le precede el carácter '/' por lo que, para hacer uso de ellas, se debe quitar.

```
var roomId;
for (roomId in io.sockets.manager.roomClients[socket.id]){
  if (roomId != '') break;
};
roomId = roomId.substring(1);
var room = io.sockets.in(roomId);
```

9. Evento de `'usuario discover'` (emitido por el cliente)
   Éste es el segundo evento definido por el programador que el código atiende. Se recibe cuando el usuario descubre una carta en su turno de juego.

```
socket.on('discover', function(card){
  var id = card.id;
  // TODO: Revisar el estado del turno del juego
});
```

10. Recuperación de un *socket* los datos asociados a un cliente
    El complemento a almacenar información en un *socket* es poder recuperarla. Al principio de la aplicación se inicializa con el nombre de usuario y la puntuación. Más tarde se podrá acceder a estas variables desde cualquier parte del código con `socket.get()` que como segundo argumento admite una función de *callback* que se invocará cuando la información esté disponible para usarse:

```
socket.get('username', function(err, username){
  // TODO:
});

socket.get('score', function(err, score){
 // TODO: actualizar puntuacion y volver a guardarla
 // TODO: informar a los jugadores de la puntuacion
});
```

11. Evento de usuario `'discover'` (emitido por el servidor)
    Este evento definido por el programador notifica a los participantes cuál

es la ruta de la imagen en el servidor que deben usar para sustituir en el navegador cuando se descubre una carta.

```
room.emit('discover',
  { id: id, src: game.cardsMap[id] }
);
```

12. Evento de usuario `'fail'`
    `'fail'` es un evento definido por el programador que se genera cuando se han descubierto dos cartas que no son iguales. Contiene la información necesaria para que el cliente pueda volver a ocultarlas: los identificadores de las cartas descubiertas que emplea la parte cliente y la ruta de la imagen con las que sustituirlas:

    ```
    room.emit('fail',
      { ids: [ lastId, id ], src: "images/back.png" }
    );
    ```

13. Evento de usuario `'success'`
    Un evento más de los definidos por el programador que indica a los participantes cuándo se ha encontrado una pareja. Envía los identificadores de cliente de las cartas de esa pareja para que la parte cliente las descarte como opciones de juego.

    ```
    room.emit('success', { ids: [lastId, id] });
    ```

14. Evento de usuario `'score'`
    `'score'` es otro de los eventos definidos por el programador, esta vez para notificar a los clientes las puntuación actualizada de uno de los participantes por lo que incluye el nombre de usuario del jugador y cuál es su puntuación actual.

    ```
    room.emit('score', { username: username, score: score } );
    ```

15. Evento de usuario `'finish'`
    El último de los eventos definidos por el programador es `'finish'`, que simplemente se emite para notificar el final del juego y que los clientes puedan cerrar las conexiones. Es un evento sin información adicional, con lo que simplemente se ejecuta

    ```
    room.emit('finish');
    ```

7.4. Objetivos de los Koans

Mediante los Koans asociados al tema de Socket.IO se persigue reforzar los puntos comentados en la Implementación con Socket.IO de la práctica. Algunos de ellos son redundantes por lo que se han seleccionado los más representativos, por ejemplo la emisión de eventos es una acción recurrente cuyas apariciones en el código son iguales, y que abarquen los aspectos principales con intención de afianzarlos. Por objetivos, éstos son:

1. Conocer que bajo el evento de servidor `'connection'` se notifica la conexión de un cliente

```
io.sockets.on( ___ , function(socket){
    socket.on(...);
    socket.on(...);
});
```

2. Definición de eventos de usuario e instalación de listeners para ellos con `on()`

```
socket.___('joins', function(message, callback){
  // TODO: Gestionar la incorporacion a un nuevo juego
});
```

3. Llamada desde el servidor a una función *callback* de asentimiento definida en el cliente

```
socket.on('joins', function(message, callback){
    var username = message.username;

    // TODO: Gestionar la incorporacion a un nuevo juego

    ___(username);
});
```

4. Conocer que con `in()` se emiten eventos en una sala concreta

```
io.sockets.___(waitingRoom).emit('start',
    { players: [startingPlayer, username] }
);
```

5. Conocer que con `emit()` se emiten eventos definidos por el programador

```
socket.___('error',
    { code: 0, extendedInfo: 'No username provided' }
);
```

6. Conocer que `roomClients` es la estructura almacena la relación entre *sockets* y *rooms*

```
var roomId;
for (roomId in io.sockets.manager.___[socket.id]){
  if (roomId != '') break;
};
```

7. Conocer que mecanismo de almacenamiento y recuperación de información en un *socket* se hace a través de `set()` y `get()`

```
socket.___('username', function(err, username){
    // TODO: Verificar que el turno corresponde al jugador
});

socket.___('score', function(err, score){
    score += 2;
    socket.___('score', score);
    room.emit('score', { username: username, score: score } );
});
```

7.5. Preparación del entorno y ejecución de los Koans

Con objeto de poder ejecutar los Koans satisfactoriamente, la práctica depende de los módulos *socket.io* y *node-uuid*.

```
$ npm install socket.io@0.9.9 node-uuid@1.3.3
```

Además, con objeto de realizar las pruebas, las dependencias que se necesita son *jasmine-node* y *socket.io-client*.

```
$ npm install socket.io-client@0.9.10 jasmine-node@1.0.26
```

Otro de los módulos que se emplean, pero que viene con la práctica, es *koanizer*,

que adapta el código para que puedan insertarse los huecos que caracterizan a un *koan*.

Si, opcionalmente, se ha completado con éxito la práctica, ésta puede ejecutarse y jugarse desde un navegador si se instala el módulo *express*.

```
$ npm install express@3.0.0beta7
```

Todos los *koans* están contenidos en el fichero `socket.io-koans.js` que es el que habrá que editar y completar, sin que quede ningún hueco ___ (tres guiones bajos seguidos).

Una vez resueltos, puede verificarse que la solución es correcta corriendo las pruebas con:

```
$ jasmine-node -verbose spec/
```

Además, como se ha comentado, una correcta resolución de la práctica la hace jugable si se ejecuta la aplicación web mediante

```
$ node app.js
```

y se apunta el navegador a la URL

`http://localhost:5555/`

7.6. Conclusión

Se concluye el capítulo después de haber alcanzado una serie de sencillas metas, cuyo fin es el de, de nuevo, conocer cómo se recibe una conexión basada en el protocolo de Socket.IO, atendiendo al evento `'connection'` y cómo el servidor puede gestionarla a través de *namespaces* o *rooms*. Otro de los aspectos que ha sido una constante a través de los capítulos es cómo se realiza el intercambio de datos ente clientes, a través de eventos que con `emit()` genera el servidor, y cómo estos clientes pueden almacenar esos datos en la propia conexión gracias al método `get()` del *socket*.

Capítulo 8

Conclusión y trabajos futuros

Llegados a este punto, se puede afirmar que se ha establecido una base bastante sólida con la que comenzar a crear aplicaciones para redes en la plataforma Node, si bien algunos tópicos avanzados han quedado fuera del libro. Sin embargo no es la finalidad abarcar todos los aspectos de Node y sus módulos, inmenso de por sí, sino fijar unos conocimientos con los que el seguir descubriendo la plataforma se convierta en algo más sencillo y rápido. No sólo a nivel de librerías sino de detalles de funcionamiento interno, indispensables para convertirse en un verdadero experto sobre ella.

No obstante, es importante resaltar que Node es una tecnología inmadura pero que evoluciona constante y rapidísimamente, por lo tanto, es bastate factible que algunos temas aquí cubiertos hayan quedado o queden pronto desactualizados. De hecho, en el momento de escribir estas líneas, Node iba por la versión 0.10.4. Cuando se comenzó el proyecto, sería la 0.6.13 la vigente. Se ha tratado por tanto de evolucionar con Node en la medida de lo posible, lo que ha llevado a que sea la versión 0.8.20 de la que finalmente se hable. Lo mismo ocurre con los módulos de terceras partes aquí explicados: Connect, Express y Socket.IO.

Queda así, el trabajo de seguir actualizando el libro, con las nuevas características y modificaciones de las actuales que se puedan producir. También es deseable introducir nuevos capítulos con módulos existentes que no se han tratado, como *tls/ssl* o *https*, o módulos que vayan surgiendo (que seguro que los habrá y en cantidad), como *domain*. Todo, por suspuesto, acompañado de su propia aplicación y diseño sugerido para ella, que sirva de aprendizaje a través de, cómo no, Koans.

Por último, se dejará todo el código generado en manos de la comunidad, que será juez imparcial de la utilidad de todo lo aquí expuesto. Si resulta aceptado, indudablemente surgirán fallos en el código que deberán ser corregidos, y sobre todo, lo más atractivo, surgirán propuestas que mejoren sustancialmente el diseño propuesto a cada uno de los problemas, con el beneficio que conlleva: aprendizaje.

Apéndice A

Listados

A continuación se incluyen los listados de todos los módulos ordenados por capítulos y completos, sin los huecos de los Koans, como referencia.

A.1. Módulos dgram y Buffer

Koans para el módulo *Buffer*, a través de la clase *RTPProtocol*:

```
var events = require('events'),
  util = require('util'),
  koanize = require('koanizer');

koanize(this);

// Standard and RFC set these values
var REFERENCE_CLOCK_FREQUENCY = 90000;

// RTP packet constants and masks
var RTP_HEADER_SIZE = 12;
var RTP_FRAGMENTATION_HEADER_SIZE = 4;

var SAMPLES_PER_FRAME = 1152; // ISO11172-3
var SAMPLING_FREQUENCY = 44100;
var TIMESTAMP_DELTA = Math.floor(SAMPLES_PER_FRAME
```

```
                                    * REFERENCE_CLOCK_FREQUENCY
                                    / SAMPLING_FREQUENCY);
var SECONDS_PER_FRAME = SAMPLES_PER_FRAME / SAMPLING_FREQUENCY;

var RTPProtocol = function(){
  events.EventEmitter.call(this);

  this.setMarker = false;
  this.ssrc = Math.floor(Math.random() * 100000);
  this.seqNum = Math.floor(Math.random() * 1000);
  this.timestamp = Math.floor(Math.random() * 1000);
};
util.inherits(RTPServer, events.EventEmitter);

RTPProtocol.prototype.pack = function(payload){

  ++this.seqNum;

  // RFC3550 says it must increase by the number of samples
  // sent in a block in case of CBR audio streaming
  this.timestamp += TIMESTAMP_DELTA;

  if (!payload) {
    // Tried to send a packet, but packet was not ready.
    // Timestamp and Sequence Number should be increased
    // anyway 'cause interval callback was called and
    // that's like sending silence
    this.setMarker = true;
    return;
  }

  var RTPPacket = new Buffer(RTP_HEADER_SIZE
                             + RTP_FRAGMENTATION_HEADER_SIZE
                             + payload.length);

  // version = 2:        10
  // padding = 0:          0
```

```
  // extension = 0:          0
  // CRSCCount = 0:           0000
/*
    KOAN #1
    should write Version, Padding, Extension and Count
*/
  RTPPacket.writeUInt8(128, 0);

  // Marker = 0:                          0
  // RFC 1890: RTP Profile for Audio and Video
  //           Conferences with Minimal Control
  // Payload = 14: (MPEG Audio Only)     0001110
  RTPPacket.writeUInt8(this.setMarker? 142 : 14, 1);
  this.setMarker = false;

  // SequenceNumber
/*
    KOAN #2
    should write Sequence Number
*/
  RTPPacket.writeUInt16BE(this.seqNum, 2);

  // Timestamp
/*
    KOAN #3
    should write Timestamp...
*/
  RTPPacket.writeUInt32BE(this.timestamp, 4);

  // SSRC
/*
    KOAN #3
    ...SSRC and...
*/
  RTPPacket.writeUInt32BE(this.ssrc, 8);

    // RFC 2250: RTP Payload Format for MPEG1/MPEG2 Video
```

```
    // 3.5 MPEG Audio-specific header
/*
    KOAN #3
    ...payload Format
*/
  RTPPacket.writeUInt32BE(0, 12);

  payload.copy(RTPPacket, 16);
  this.emit('packet', RTPPacket);
};

module.exports = exports.RTPProtocol = RTPProtocol;
```

Koans para el módulo *dgram*, a través de la clase *Sender*:

```
var dgram = require('dgram'),
  koanize = require('koanizer');

koanize(this);

var Sender = function(options){
  var options = options || {}
  this.port = options.port || 5000;
  this.broadcastAddress = options.broadcastAddress || '224.0.0.14';
  this.stats = {
      txPackets : 0,
      txBytes : 0
  };
/*
    KOAN #1
    should create udp sockets properly
*/
  this.txSocket = dgram.createSocket('udp4');
  this.rxSocket = dgram.createSocket('udp4');
};

Sender.prototype.start = function(){
/*
```

```
    KOAN #2
    should make udp server listening sucessfully
*/
  this.rxSocket.bind(5001);
};

Sender.prototype.broadcast = function(packet){
  var self = this;
/*
    KOAN #3
    should send a message correctly
*/
  this.txSocket.send(packet,
                     0,
                     packet.length,
                     this.port,
                     this.broadcastAddress,
                     function(err, bytes){
      ++self.stats.txPackets;
      self.stats.txBytes += bytes;
  });
};

Sender.prototype.enableStats = function(enable){
  var self = this;
  if (enable){
/*
    KOAN #4
    should attend incoming packets from clients
*/
    this.rxSocket.on('message', function(msg, rinfo){
        var stats = new Buffer(JSON.stringify(self.stats));
/*
    KOAN #5
    should response to clients with stats messages
*/
        dgram.createSocket('udp4').send(stats,
```

```
                                                0,
                                                stats.length,
                                                5002,
                                                rinfo.address);
      })
  }else{
    this.rxSocket.removeAllListeners();
  }
};

Sender.prototype.end = function(){
  this.rxSocket.close();
  this.txSocket.close();
}

exports.Sender = Sender;
```

A.2. Módulos *net* y *Stream*

Koans para los módulos *net* y *Stream*, a través de la clase *RemotePrompt*:

```
var mediaLibraries = require('./Mp3Library'),
  sources = require('./Mp3Source'),
  RTPServer = require('./RTPServer'),
  udp = require('dgram'),
  net = require('net'),
  koanize = require('koanizer');

koanize(this);

var RemotePrompt = function(library){

  var sessionsDB = {};
/*
    KOAN #1
    should instantiate a TCP Server
```

```
*/
  this.server = net.createServer();
  this.listen = function(port){
/*
    KOAN #2
    should be able to listen to incoming connections
*/
    this.server.listen(port);
  };

  this.close = function(){
    this.server.close();
  };

/*
    KOAN #3
    should attend incoming connections
*/
  this.server.on('connection', function(connection){
    var remoteIP = connection.remoteAddress;
/*
    KOAN #4
    should write in connection socket
*/
    connection.write("Welcome to your command line playlist manager, "
                     + remoteIP);

    if (remoteIP in sessionsDB){
/*
    KOAN #5
    should be able to close connections
*/
      connection.end("Duplicated session, closing.");
      return;
    };

    sessionsDB[remoteIP] = true;
```

```
    var source = new sources.Mp3Source(library);
    var rtpserver = new RTPServer();
    var udpSocket = udp.createSocket('udp4');

    rtpserver.on('packet', function(packet){
      udpSocket.send(packet, 0, packet.length, 5002, remoteIP);
    });

    source.on('frame', function(frame){
      rtpserver.pack(frame);
    });

    source.on('track', function(trackName){
      connection.write("Now playing " + trackName + "\r\n# ");
    });

    source.on('listEnd', function(){
      var seconds = 10;
      connection.write("End of the list reached.Closing in "
                       + seconds
                       + " seconds\r\n# ");
/*
    KOAN #6
    should trigger a unactivity timeout on the socket
*/
      connection.setTimeout(seconds * 1000, function(){
        delete sessionsDB[this.remoteAddress];
        connection.end("Your session has expired. Closing.");
      });
    });
/*
    KOAN #7
    should receive incoming data from connections
*/
    connection.on('data', function(data){
```

```
  // disable timeout, in case it was set
  this.setTimeout(0);

  var command = data.toString('utf8').split("\r\n")[0];
  switch(command){
    case "list":
      var playlist = source.list();
      this.write("\r\nSongs in the playlist");
      this.write("\r\n---------------------");
      for (var i=0; i < playlist.length; i++){
        var song = playlist[i];
        this.write("\r\n"
                   + (source.currentTrack() == song? "> " : "  ")
                   + song);
      }
      this.write("\r\n# ");
      break;
    case "play":
      source.play();
      break;
    case "pause":
      source.pause();
      break;
    case "next":
      source.next();
      break;
    case "prev":
      source.prev();
      break;
    case "exit":
      delete sessionsDB[this.remoteAddress];
      this.end("Bye.");
      break;
    default:
      this.write("Command " + command + " unknown\r\n# ");
  }
});
```

```
    connection.on('close', function(){
      source.pause();
      udpSocket.close();
      rtpserver = source = null;
    });

    connection.write("\r\nNow, point your player to:\r\n\trtp://"
                     + remoteIP
                     + ":5002\r\n# ");
  });
};

exports.create = function(){
  var actions = [];

  var app;
  var library = new mediaLibraries.Mp3Library;
  library.on('ready', function(){
    app = new RemotePrompt(this);

    for (var i = 0; i< actions.length; i++){
      actions[i].apply(app);
    };
    actions = undefined;
  });

  // TO COMPLETELY IGNORE:
  // some kind of ugly, lame & messy code mixing promise pattern
  //  with proxy pattern.
  //
  // This offers an object with same listen method as RemotePrompt
  // class invocable when app is not ready yet. It offers too an
  // onListening method that will install a callback on 'listening'
  // event on server property of RemotePrompt; this method should
  // not be in RemotePrompt class itself but it's useful for
  // testing purposes. It's also chainable.
```

```
  return new function(){
    var _defer = function(callback){
      if (actions){
        actions.push(callback);
      } else {
        callback.apply(app)
      }
    };

    var self = this;
    this.listen = function(port){
      _defer(function(){
        this.listen(port);
      })
    };

    this.close = function(){
      _defer(function(){
        this.close();
      })
    };

    this.onListening = function(callback){
      _defer(function(){
        this.server.on('listening', callback);
      });
      return self;
    }
  }
};
```

A.3. Módulo *http*

Koans para el módulo *http* a través del objeto *listServer*

```
var http = require('http'),
  url = require('url'),
  fs = require('fs'),
  querystring = require('querystring'),
  koanize = require('koanizer');

koanize(this);

var listServer = http.createServer();
/*
    KOAN #1
    should make the server to response incoming requests
*/
listServer.on('request', function(req, res){

  var self = this;
/*
    KOAN #2
    should make the server to properly read the requests headers
*/
  var credentials = req.headers["authorization"];
  var userAndPass, broadcastIp, pass;
  if (credentials){
    userAndPass = new Buffer(credentials.split(' ')[1], 'base64')
                  .toString('ascii')
                  .split(':');
    broadcastIp = userAndPass[0];
    pass = userAndPass[1];
  }
  if (!credentials
      || !(broadcastIp in allowedList
           && broadcastIp in this.broadcastList)
      || allowedList[broadcastIp] != pass) {
    res.writeHead(401, "Unauthorized", {
      "WWW-Authenticate": 'Basic realm="List Server Authentication"'
    });
    res.end();
```

```
    return;
  }
/*
    KOAN #3
    should make the server to use url module
*/
  var uri = url.parse(req.url);
/*
    KOAN #4
    should make the server to identify the request method
*/
  switch(req.method){
    case 'GET':
      var path = uri.pathname;
      if ( path == '/' ){
        var player = this.broadcastList[broadcastIp];
        writeDocument(res, {group:broadcastIp,
                            paused:player.paused,
                            tracks:player.list(),
                            currentTrack:player.currentTrack()});
      } else {
        fs.readFile("." + path, function(error, data){
          if (error){
            res.writeHead(404);
            res.end();
            return;
          }

          var fileExtension = path.substr(path.lastIndexOf(".")+1);
          var mimeType = MIME_TYPES[fileExtension];
/*
    KOAN #5
    should make the server to include new header in
    implicit requests
*/
          res.setHeader("Content-Type", mimeType);
          if (mimeType.indexOf("text/") >= 0){
```

```
            res.setHeader("Content-Encoding", "utf-8");
          }
          res.setHeader("Content-Length", data.length);
          res.write(data, "binary");
          res.end();
        });
      }
      break;

    case 'POST':
      req.setEncoding('utf8');

      var body = '';
      req.on('data', function(data){
        body += data;
      })

      req.on('end', function(){
/*
    KOAN #6
    should make the server to use querystring module
*/
        var query = querystring.parse(body);
        var action = query.action;

        var player = self.broadcastList[broadcastIp];
        if (action in player) {
          player[action](function(){
            writeDocument(res, {group:broadcastIp,
                                paused:player.paused,
                                tracks:player.list(),
                                currentTrack:player.currentTrack()});
          });
        } else {
            res.writeHead(500, "Internal Server Error");
            res.end();
        }
```

```
      });
      break;

    default:
      res.writeHead(501, "Not Implemented");
      res.end();
      break;
  }
});

var writeDocument = function(res, doc){
  var head = '<html><head>'
             +'<link rel="stylesheet"'
             +' type="text/css"'
             +' href="/static/style.css">'
             +'</head><body>';
  var tail = '</body></html>';
  var info = '<p>Playlist for group ' + doc.group + '</p>';
  var form = '<form method="post" action="/">';
  form += '<input type="submit" value="prev" name="action">';
  form += doc.paused?
          '<input type="submit" value="play" name="action">'
          :'<input type="submit" value="pause" name="action">'
  form += '<input type="submit" value="next" name="action"></form>'

  var trackList = doc.tracks;

  var list = "<ul>";
  for(var i=0, l=trackList.length; i<l; i++){
    var track = trackList[i];
    list += ('<li'
             + (track == doc.currentTrack?
                         ' class="currentTrack"'
                         : '')
             + '>'
             + track
             + '</li>');
```

```
  };
  list += "</ul>";

  var content = head + info + form + list + tail;
  res.writeHead(200, 'OK', {
    "Content-type": "text/html",
    "Content-length" : content.length
  });
  res.write(content);
  res.end();
}

var allowedList = {
  "224.0.0.114": "password"
}

var MIME_TYPES = {
  "png" : "image/png",
  "css" : "text/css",
  "js" : "text/javascript",
  "html" : "text/html"
}

exports.create = function(db){
  if (db == null) throw new Error('Database cannot be empty');

  listServer.broadcastList = db;
  return listServer;
}
```

A.4. Módulo *Express*

Koans para el objetivo nº 1 del módulo *Express*

```
var express = require('express'),
  crypto = require('crypto'),
```

```
  koanize = require('koanizer');

koanize(this);

var models = require('./models/whizr.js');

var app = express();

app.engine('.html', require('ejs').__express);
app.set('view engine', 'ejs');

app.configure(function(){
  /*
    KOAN #1
    The Application must be properly
    configured to serve favicon
  */
  app.use(express.favicon()); // avoid call twice the routes
  app.use(express.static(__dirname + '/public'));
  app.use(express.cookieParser());
  app.use(express.session({ secret: "Node.js koans" }));
  app.use(express.bodyParser()); // to parse post params
});

/**********************
  PROFILE & HOME     *
***********************/

/*
    KOAN #2
    Application must handle index page
*/
app.get('/', function(req, res){
  res.sendfile('./views/index.html');
});

app.get('/:username', function(req, res){
```

```
  var username = req.param('username');
  models.Whizr.findOne({ username: username },
                       function(err, doc){
    if (err || doc == null) {
/*
  KOAN #3
  Application must be able to generate simple responses
*/
      res.send(404, 'Not found');
    } else {
/*
  KOAN #4
  Application must be able to produce dynamic responses
  according to a view
*/
        res.render('home.html', { name: doc.name,
                                  username: doc.username,
                                  whizr: req.session.whizr } )
        }
  });
});

app.get('/:username/profilepic', function(req, res){
  models.Whizr.findOne({ username: req.params.username },
                       function(err, doc){
    if(err){
      res.send('Not Found', 404);
    } else {
      res.redirect("http://www.gravatar.com/avatar/"
                   + doc.emailHash
                   + "?s=100");
    }
  });
});

/*****************************
  REGISTER & LOGIN & LOGOUT *
```

```
*****************************/

/*
    KOAN #5
    Application must handle action endpoints for forms
*/
app.post('/login', function(req, res){

  //In case we're logged in
  if (req.session.whizr != undefined) {
    res.redirect('/' + req.session.whizr.username);
  }

  models.Whizr.findOne({ username: req.param('username') },
                       function(err, doc){
    if (err) {
      res.send('Error', 500);
            return;
    }

    if ( doc.password == req.param('password') ) {
      req.session.whizr = doc;
      res.redirect('/' + doc.username);
    } else {
      res.send('Unauthorized', 401);
    };
  } )

});

app.post('/logout', function(req, res){
  if (req.session.whizr) {
    var redirection = req.session.whizr.username;
    req.session.destroy();
    res.redirect('/' + redirection);
  } else {
    res.send('Unauthorized', 401);
```

```
  }
});

app.post('/register', function(req, res){
  var username = req.param('username');
  var password = req.param('password');
  var name = req.param('name');
  var email = req.param('email');

  if ( username.length <= 4
       || username.length >= 10
       || password.length <= 4
       || password.length >= 10) {
    res.redirect(400, '/'); // error
        return;
  }

  models.Whizr.findOne({ username: username }, function(err, doc){
    if (err || doc != null) {
      res.send('Error', 500);
      return;
    }

    var whizr = new models.Whizr();
    whizr.name = name;
    whizr.username = username;
    whizr.password = password;
    whizr.email = email;
    var hash = crypto.createHash('md5');
    hash.update(email);
    whizr.emailHash = hash.digest('hex');
    whizr.newMentions = 0;

    whizr.save(function(err){
      if (err) {
          res.send('Error', 500);
      };
```

```
      req.session.whizr = whizr;
      res.redirect('/' + username);
    });
  });
});

module.exports = exports = app;
```

Koans para el objetivo nº 2 del módulo *Express*

```
var express = require('express'),
  crypto = require('crypto'),
  koanize = require('koanizer');

koanize(this);

var models = require('./models/whizr.js');

var app = express();

app.engine('.html', require('ejs').__express);
app.set('view engine', 'ejs');

app.configure(function(){
  app.use(express.favicon()); // avoid call twice the routes
  app.use(express.static(__dirname + '/public'));
  app.use(express.cookieParser());
  app.use(express.session({ secret: "Node.js koans" }));
  app.use(express.bodyParser()); // to parse post params
});

/**********************
  INDEX & PROFILE    *
**********************/

app.get('/', function(req, res){
  res.sendfile('./views/index.html');
});
```

```
app.get('/:username', function(req, res){

  var username = req.params('username');

  models.Whizr.findOne({ username: username },
                       function(err, doc){
    if (err || doc == null) {
      res.send('Not found', 404);
            return;
    }

    models.Whiz.find({ author: username },
                     function(err, docs){
      if (err || doc == null) {
        res.send('Not found', 404);
      }
      res.render('home.html', { name: doc.name,
                                username: doc.username,
                                whizs: docs,
                                whizr: req.session.whizr });
    } ).sort({ date: -1 });

  });
});

app.get('/:username/profilepic', function(req, res){
  models.Whizr.findOne({ username: req.params.username },
                       function(err, doc){
    if(err){
      res.send('Not Found', 404);
    };
    res.redirect("http://www.gravatar.com/avatar/"
                 + doc.emailHash
                 + "?s=100");
  });
});
```

```
/*******************************
  REGISTER & LOGIN & LOGOUT *
*******************************/

app.post('/login', function(req, res){

  //In case we're logged in
  if (req.session.whizr != undefined) {
    res.redirect('/' + req.session.whizr.username);
  }

  models.Whizr.findOne({ username: req.param('username') },
                       function(err, doc){
    if (err) {
      res.send('Error', 500);
    }

    if ( doc.password == req.param('password') ) {
      req.session.whizr = doc;
      res.redirect('/' + doc.username);
    } else {
      res.send('Unauthorized', 401);
    };
  })
});

// Filter to check authentication in routes that could need it
var checkAuth = function(req, res, next){
/*
    KOAN #1
    Filter should be chainable to others
*/
  req.session.whizr? next() : res.send('Unauthorized', 401);
}

/*
```

```
    KOAN #2
    Application must allow or deny access to certain endpoints
*/
app.post('/logout', checkAuth, function(req, res){
    var redirect = req.session.whizr.username;
    req.session.destroy();
    res.redirect('/' + redirect);
});

app.post('/register', function(req, res){
  var username = req.param('username');
  var password = req.param('password');
  var name = req.param('name');
  var email = req.param('email');

  if ( username.length <= 4
       || username.length >= 10
       || password.length <= 4
       || password.length >= 10) {
    res.redirect('/'); // error
        return;
  }

  models.Whizr.findOne({ username: username },
                       function(err, doc){
    if (err || doc != null) {
      res.send('Error', 500);
      return;
    }
    var whizr = new models.Whizr();
    whizr.name = name;
    whizr.username = username;
    whizr.password = password;
    whizr.email = email;
    var hash = crypto.createHash('md5');
    hash.update(email);
    whizr.emailHash = hash.digest('hex');
```

```
    whizr.newMentions = 0;

    whizr.save(function(err){
      if (err) {
        res.send('Error', 500);
        return;
      }
      req.session.whizr = whizr;
      res.redirect('/' + username);
    });
  });
});


/*************
  WHIZEAR  *
*************/

app.post('/whizr', checkAuth, function(req, res){
  var text = req.param('whiz');

  if ( text == null
       || text.length  == 0
       || text.length >= 140) {
    res.redirect('Error', 404);
    return;
  }

  var whiz = new models.Whiz();
  whiz.text = text;
  whiz.author = req.session.whizr.username;
  whiz.save(function(err){
    if (err) {
      res.send('Error', 500);
      return;
    }
    res.redirect('/' + req.session.whizr.username);
```

```
  });
});

/**********************
   FOLLOW & UNFOLLOW *
**********************/

app.post('/follow', checkAuth, function(req, res){
  var followTo = req.param('username');

  if (followTo.length == 0
      || followTo == null
      || followTo == req.session.whizr.username){
    res.send('Error', 500);
    return;
  }
/*
    KOAN #3
    Application must be able to update user's profiles
*/
  models.Whizr.update( { username: req.session.whizr.username },
                       { $addToSet: { following: followTo } },
                       null,
                       function(err, numAffected){
    if (!err) {
      req.session.whizr.following.push(followTo);
      console.log(req.session.whizr.following);
      res.redirect('/' + followTo);
    }
  });
});

app.post('/unfollow', checkAuth, function(req, res){
  var unfollow = req.param('username');
  if (unfollow.length == 0 || unfollow == null || unfollow == whizr){
    res.send('Error', 500);
    return;
```

```
  }
/*
    KOAN #3
    Application must be able to update user's profiles
    according to certain conditions
*/
  models.Whizr.update( { username: req.session.whizr.username },
                       { $pull: { following: unfollow } },
                       null,
                       function(err, numAffected){
    if (!err) {
      // update the session
      var following = req.session.whizr.following;
      following.splice(following.indexOf(unfollow), 1);
      res.redirect('/' + unfollow);
    }
  });
});

module.exports = exports = app;

// To ignore: for testing purposes
app.get('/:username/following', function(req, res){
    models.Whizr.findOne({username: req.param('username')},
                         function(err, doc){
      if(err){
        res.send('Not Found', 404);
      };
      res.send(200, { following: doc.following });
    });
});
```

A.5. Módulo *Socket.IO*

Koans para el módulo *Socket.IO*:

```
var socketio = require('socket.io'),
  uuid = require('node-uuid'),
  Game = require('./models/Game.js'),
  util = require('util'),
  koanize = require('koanizer');

koanize(this);

var room2game = {},
  waitingRoom = null,
  startingPlayer = null,
  io = null;

exports.createGame = function(server){
  io = socketio.listen(server);
  io.set('log level', 1);

/*
  KOAN #1
  Server must be able to receive incoming connections
*/
  io.sockets.on('connection', function(socket){
/*
  KOAN #2
  The server must be able to properly
  act to joins messages from client
*/
    socket.on('joins', function(message, callback){
      var username = message.username;
      if (username
          && username != ''
          && startingPlayer != username){
        socket.set('username', username);
        socket.set('score', 0);
/*
  KOAN #3
  As result of the joins message, the Server must acknowledge
```

```
  it sending the username back to the client
*/
        callback(username);

        if (!waitingRoom){
          startingPlayer = username;
          waitingRoom = uuid.v1();
          room2game[waitingRoom] = new Game();
          socket.join(waitingRoom);
        } else {
          room2game[waitingRoom].lastTurn = username;
          socket.join(waitingRoom);
/*
  KOAN #4
  Having two players in a room, the server must be able to
  notify both the start of the game
*/
          io.sockets.in(waitingRoom)
            .emit('start',
                  {players: [startingPlayer, username]});
          waitingRoom = null;
          startingPlayer = null;
        };
      } else {
/*
  KOAN #5
  The server must handle properly faulty inputs
*/
        socket.emit('error',
                    { code: 0,
                      msg: 'Invalid username' });
      }
    });

    socket.on('discover', function(card){
      var socket = this;
      var id = card.id;
```

```
      var roomId = '';
/*
  KOAN #6
  The server must be able to know what room the client is in
*/
      for (roomId in io.sockets.manager.roomClients[socket.id]){
        if (roomId != '') break;
      };
      roomId = roomId.substring(1);
      var room = io.sockets.in(roomId);
      var game = room2game[ roomId ];

      if (game === undefined || !(id in game.cardsMap)) {
        return;
      }

/*
  KOAN #7 (I)
  The socket must obtain any client info from the socket
*/
      socket.get('username', function(err, username){

        if (err) {
          console.log("Discover: username error", err);
          process.exit();
        }

        if (game.lastTurn != username){

          room.emit('discover',{id: id, src: game.cardsMap[id]})

          var lastId = game.lastCard;
          if (lastId == null || lastId == id) {
            game.lastCard = id;
            return;
          };
```

```
          if (game.cardsMap[lastId] == game.cardsMap[id]){

            delete game.cardsMap[lastId];
            delete game.cardsMap[id];
            game.lastCard = null;

            room.emit('success', { ids: [lastId, id] });
/*
  KOAN #7 (II)
  The socket must obtain any client info from the socket
  and, once updated, save in it again
*/
            socket.get('score', function(err, score){
              score += 2;
              socket.set('score', score);
              room.emit('score',
                        {username: username, score: score});
            });
          } else {
            game.lastTurn = username;
            game.lastCard = null;
            room.emit('fail', {ids: [ lastId, id ],
                               src: "images/back.png"});
          };

          if (game.isOver()) {
            room.emit('finish');
            delete room2game[roomId];
          }
        }
      });
    });

    // To ignore: testing purposes
    socket.emit('connectionDone');
  });
  return server;
```

```
}

exports.endGame = function(){
  if (io) {
    io.server.close();
  }
}
```

Bibliografía

[1] Ryan dahl, July 2010. http://www.youtube.com/watch?v=F6k8lTrAE2g.

[2] Guillermo Rauch. Socket.IO client. https://github.com/LearnBoost/socket.io-client.

[3] Oleg Podsechin. Ryan dahl interview. http://dailyjs.com/2010/08/10/ryan-dahl-interview/.

[4] Node.JS. http://nodejs.org.

[5] POSIX Austin Joint Working Group. Portable operating system interface (posix(r)). http://standards.ieee.org/findstds/standard/1003.1-2008.html.

[6] Core modules. http://nodejs.org/api/modules.html#modules_core_modules.

[7] Kris Kowal. Commonjs effort sets javascript on path for world domination. http://arstechnica.com/web/news/2009/12/commonjs-effort-sets-javascript-on-path-for-world-domination.ars.

[8] Modules/1.1.1. http://wiki.commonjs.org/wiki/Modules/1.1.1.

[9] Isaac Z. Schlueter. Node v0.8.0. http://blog.nodejs.org/2012/06/25/node-v0-8-0/.

[10] A Costello. Rfc 3492-punycode: A bootstring encoding of unicode for internationalized domain names in applications (idna). *Network Working Group. Disponwel na Internet em http://www. ietf. org/rfc/rfc3492. txt*, 2003.

[11] Marc Lehmann. libev - a high performance full-featured event loop written in c. http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod.

[12] ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, fifth edition, June 2011.

[13] W3C DOM Working Group et al. Document object model, 2002.

[14] David Flanagan. *JavaScript: the definitive guide*. O'Reilly Media, Incorporated, 2006.

[15] Functional programming. http://www.haskell.org/haskellwiki/Functional_programming.

[16] Tim Caswell. Why use çlosure¿ http://howtonode.org/why-use-closure.

[17] Cade Metz. The node ahead: Javascript leaps from browser into future. http://www.theregister.co.uk/2011/03/01/the_rise_and_rise_of_node_dot_js/.

[18] Octane javascript benchmark. http://octane-benchmark.googlecode.com/svn/latest/index.html.

[19] Licencia new bsd o bsd de 3 cláusulas. http://opensource.org/licenses/BSD-3-Clause.

[20] Google. Chrome v8 introduction. https://developers.google.com/v8/intro.

[21] Garbage collection. http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29#Stop-the-world_vs._incremental_vs._concurrent.

[22] Google. V8 embedder's guide. https://developers.google.com/v8/embed.

[23] Felix Geisendörfer. The node.js scalability myth. https://speakerdeck.com/felixge/the-nodejs-scalability-myth.

[24] Microsoft. Designing for scalability. http://msdn.microsoft.com/en-us/library/aa291873(v=vs.71).aspx.

[25] Alex Payne. Node and scaling in the small vs. scaling in the large, July 2010. http://al3x.net/2010/07/27/node.html.

[26] Ryan Dahl. Ryan dahl: Introduction to node.js. www.youtube.com/watch?v=M-sc73Y-zQA.

[27] libuv. https://github.com/joyent/libuv.

[28] Ryan Dahl. libuv status report. http://blog.nodejs.org/2011/09/23/libuv-status-report/.

[29] Mikito Takada. Understanding the node.js event loop. http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/.

[30] Marc Lehmann. libeio - truly asynchronous posix i/o. http://http://pod.tst.eu/http://cvs.schmorp.de/libeio/eio.pod.

[31] Isaac Z. Schlueter. How to module. http://howtonode.org/how-to-module.

[32] Isaac Schlueter. Which callbacks are sent to the event loop? https://groups.google.com/d/topic/nodejs/qluCAFK5zp4/discussion.

[33] Juan Antonio de la Puente. Introducción a los sistemas de tiempo real, 2007. http://laurel.datsi.fi.upm.es/~ssoo/STR/Introduccion.pdf.

[34] Brian Cantrill. Instrumenting the real-time web: Node.js in production. http://www.slideshare.net/bcantrill/instrumenting-the-realtime-web-nodejs-in-production.

[35] Felix Geisendörfer. Convincing the boss. http://nodeguide.com/convincing_the_boss.html.

[36] Christopher Kent, Jeffrey C Mogul, et al. *Fragmentation considered harmful*, volume 17. 1987.

[37] Z Albanna, K Almeroth, D Meyer, and M Schipper. Rfc3171: Iana guidelines for ipv4 multicast address assignments. Technical report, Technical report, Internet Engineering Task Force (IETF), 2001.

[38] Ascii. http://en.wikipedia.org/wiki/ASCII.

[39] Joel Spolsky. The absolute minimum every software developer absolutely, positively must know about unicode and character sets. http://www.joelonsoftware.com/articles/Unicode.html.

[40] David H Crocker. Rfc 822, standard for the format of arpa internet text messages, august 1982. *URL http://www. cis. ohio-state. edu/htbin/rfc/rfc822. html*, 23:38–41.

[41] Network Working Group et al. Mime (multipurpose internet mail extension). Technical report, RFC 1341, avril, 1993.

[42] H SHULZRINNE, S CASNER, R FREDERICK, et al. Rfc3350. *RTP: a transport protocol for real time application*, 2003.

[43] S Casner and H Schulzrinne. Rfc 3551: Rtp profile for audio and video conferences with minimal control. Technical report, Technical report, Columbia University, Packet Design, 2003.

[44] D Hoffman, G Fernando, V Goyal, and M Civanlar. Rfc2250: Rtp payload format for mpeg1. *MPEG2 video*, 1998.

[45] ISO Iso. IEC 11172-3 Information technology-coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s-Part3: Audio. *Motion Picture Experts Group*, 1993.

[46] Andrew Tanenbaum. *Computer networks*. Prentice Hall Professional Technical Reference, 2002.

[47] Jon Postel. Rfc 793: Transmission control protocol, september 1981. *Status: Standard*, 2003.

[48] R Braden. Rfc 1122: Requirements for internet hosts—communication layers. *Status: Standard*, 1989.

[49] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. Rfc 2616: Hypertext transfer protocol–http/1.1. *Network Working Group and others*, 1999.

[50] TJ Holowaychuck. Connect - Middleware For NodeJS. http://tjholowaychuk.com/post/664516126/connect-middleware-for-nodejs.

[51] Cross-site request forgery (csrf). https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

[52] Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.

[53] Matt Bridges. application/x-www-form-urlencoded or multipart/form-data? http://stackoverflow.com/questions/4007969/application-x-www-form-urlencoded-or-multipart-form-data.

[54] What are "signed" cookies in connect/expressjs? http://stackoverflow.com/questions/11897965/what-are-signed-cookies-in-connect-expressjs.

[55] James Allen. Cookie sessions. https://github.com/jpallen/connect-cookie-session.

[56] Name-based virtual host support. http://httpd.apache.org/docs/2.0/en/vhosts/name-based.html.

[57] Ricky Ho. Mongodb architecture. http://horicky.blogspot.it/2012/04/mongodb-architecture.html.

[58] Amy Brown and Greg Wilson. The architecture of open source applications. *Lulu. com*, 2011.

[59] Tony Hannan. Why mongodb? [http://www.mongodb.org/display/DOCS/Introduction.

[60] Introduction to mongodb. http://www.mongodb.org/display/DOCS/Philosophy.

[61] Guillermo Rauch. Socket.io: the crossbrowser websocket for real-time apps. http://socket.io.

[62] Guillermo Rauch. Socket.io, frequently askes questions. http://socket.io/#faq.

[63] I Fette and A Melnikov. Rfc 6455: The websocket protocol. Technical report, Status: Internet Draft. Available at: http://tools. ietf. org/html/draft-ietf-hybi-thewebsocketprotocol-17. Accessed 8-February-2012, 2011.

[64] Ian Hickson. The websocket api. *W3C Working Draft WD-websockets-20110929, September*, 2011.

[65] Tian Davis. Simple long polling example with javascript and jquery. http://techoctave.com/c7/posts/60-simple-long-polling-example-with-javascript-and-jquery.

[66] Comet. http://en.wikipedia.org/wiki/Comet_%28programming%29.

[67] Viacheslav Tykhanovskyi. Socket.io for backend developers. http://showmetheco.de/articles/2011/8/socket-io-for-backend-developers.html.

[68] Kyle Simpson. JSONP: Safer cross-domain AJAX. http://www.json-p.org/.

[69] Jerod Venema. What is JSONP all about? http://stackoverflow.com/questions/2067472/please-explain-jsonp.

[70] Jsonp, how it works. http://en.wikipedia.org/wiki/JSONP#How_it_works.

[71] Bob Ippolito. Remote JSON - JSONP. http://bob.ippoli.to/archives/2005/12/05/remote-json-jsonp/.

[72] Authorizing. https://github.com/LearnBoost/socket.io/wiki/Authorizing.

[73] Exposed events. https://github.com/LearnBoost/socket.io/wiki/Exposed-events.

[74] Rooms. https://github.com/LearnBoost/socket.io/wiki/Rooms.

[75] Eugene Beresovksy. Socket.io rooms or namespacing? http://stackoverflow.com/questions/10930286/socket-io-rooms-or-namespacing.

[76] Peleus Uhley. Setting up a socket policy file server. http://www.adobe.com/devnet/flashplayer/articles/socket_policy_files.html.

[77] Guillermo Rauch. Socket.IO protocol. https://github.com/LearnBoost/socket.io-spec#handshake.