UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

# NODE.JS Do's and Don'ts

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor**: Juan Íscar Martínez

**Tutor**: Vicente Pelechano Ferragud

2014-2015

# Resumen

Node.js es una plataforma construida con el intérprete de JavaScript de Chrome para construir fácilmente aplicaciones rápidas de red escalables. Node.js es un modelo no bloqueante de E/S basado en eventos, que lo hace ligero y eficiente, ideal para aplicaciones en tiempo real de datos intensivos que se ejecutan a través de dispositivos distribuidos. Este entorno ha adquirido una gran importancia hoy en día y está siendo usado en gran cantidad de dominios de aplicación. El trabajo fin de grado tiene como objetivo que el alumno que estudia en qué ámbitos es ideal su aplicación y en qué ámbitos o tipos de aplicaciones no es recomendable utilizarlo tenga una mejor referencia. El alumno va a desarrollar casos reales para demostrar su idoneidad.

**Palabras clave:** Javascript, Servicios Web, REST, Servidores Web.

# Abstract

Node.js is a programming environment on the server layer based on the Javascript programming language. It was created with the approach to be helpful in creating highly scalable network programs. This environment has become very important today and is being used in many application domains. The final project aims that students study in which areas is an ideal application and in what areas or types of applications is not advisable to use it to have a better reference. The student will develop case studies to demonstrate their qualifications.

**Keywords :** Javascript, Web Services , REST, Web Servers.

# Tabla de contenidos

## Contenido

# Tabla de ilustraciones

## Ilustraciones

# 1. Introducción

## 1.1. Motivación

Existen multitud de servidores disponibles (Web, SMTP, Proxies, etc), permitiendo escoger el servidor que mejor se adapte a las necesidades del sistema que se pretende diseñar. En algunos casos, estos servidores no cumplen los requisitos exigidos, por lo que es necesario el desarrollo de un servidor que se adapte a la aplicación.

Node.js surge como opción para desarrollar servidores según las necesidades del sistema, de cualquier tipo y fácilmente desde cero. Los servidores construidos con Node.js, ofrecen unas propiedades muy diferentes a las de otros servidores implementados con tecnologías convencionales, convirtiéndolo en una alternativa idónea para usos como el Streaming y otras aplicaciones que necesiten gestionar gran cantidad comunicaciones. Igualmente, este framework tiene una serie de desventajas para otros usos tales como cómputo en general.

Este trabajo de fin de grado, tiene como objetivo servir de guía en la decisión de si se debe utilizar Node.js para el desarrollo de un sistema o aplicación concreta, detallando tanto los aspectos positivos de esta tecnología como de las desventajas que ofrece, y por último se mostraran ejemplos de servidores construidos con Node.js explicando su funcionamiento.

## 1.2. Objetivos del Proyecto

El principal objetivo de este proyecto es ayudar al desarrollador sin conocimientos iniciales de Node.js, a tomar decisiones y/o a determinar si es una solución tecnológica adecuada para su sistema, indicando tanto las propiedades como sus inconvenientes, su funcionamiento y contrastando dichas propiedades con otros lenguajes y frameworks.

Estas conclusiones se han obtenido analizando las características de Node.js, los servidores en funcionamiento, en pruebas realizadas y en el conocimiento extraído analizando los resultados.

Los objetivos son:

**1. Analizar las propiedades y funcionamiento de Node.js**

Enumerando todo lo relativo a sus propiedades.

**2. Analizar proyectos desarrollados.**

Buscando aplicaciones construidas en Node.js, analizando su funcionamiento y el motivo de su idoneidad en caso de ser favorable.

**3. Identificar proyectos y aplicaciones típicas.**

Una vez se han analizado los programas en funcionamiento, se mostraran qué tipo de aplicaciones son adecuadas.

**4. Orientar en la elección de Node.js.**

Cuando las propiedades ya han sido enumeradas, es momento de elegir si encajan con las necesidades de nuestro sistema, y si algunos aspectos de Node.js presentan problemas con la aplicación, se ofrecerán soluciones y alternativas.

## 1.3. Estructura

La memoria está estructurada en los siguientes capítulos:

**1. Introducción:** Se encuentra divido en tres apartados: Introducción, objetivos y motivación del proyecto.

**2. Node.js. Una revisión técnica:** Este capítulo presenta el funcionamiento de Node.js, explicando su origen, con qué objetivos se ha desarrollado y sus propiedades más básicas.

**4. Ventajas del uso de Node.js:** Capitulo destinado a enumerar y explicar las propiedades de Node.js.

**5. Tipos de Aplicaciones Recomendadas:** Se enumeraran los tipos de aplicaciones que por su naturaleza cumplen con las propiedades de Node.js.

**6. Inconvenientes del uso de Node.js:** Capitulo destinado a enumerar y explicar las desventajas más significativas.

**7. Tipos de Aplicaciones Problemáticas:** Se enumeraran los tipos de aplicaciones que no encajan con la filosofía de diseño de Node.js.

**8. Análisis de aplicaciones en uso:** Se seleccionaran programas en uso y se comparan con las propiedades y los tipos de aplicaciones antes descritas, indicando si utilizar Node.js ha sido una elección adecuada.

**9. Un Marco Conceptual para evaluar la Idoneidad del uso de Node.Js:** Capítulo en el que se ayudará a decidir si usar Node.js basándose en las propiedades y comparaciones anteriores con las necesidades del sistema en el que se implementará.

**10. Validación de la Propuesta:** En el capítulo se expondrán análisis de aplicaciones en uso y de una desarrollada para el proyecto (la cual llamaremos

**Node-censo**), explicando las fases del desarrollo, toma de decisiones y conclusiones del programa resultado.

**11. Comparación y Evidencias:** Se compara la aplicación construida (**Node-censo**) con las anteriores aplicaciones analizadas y se propondrán mejoras.

**12. Conclusiones:** Capitulo resumen de los datos más significativos del proyecto, así como de posibilidades de mejora.

# 2. Node.js. Una revisión técnica

## 2.1. Introducción a Node.js

Node.js fue creado por Ryan Dahl en 2009. La idea inicial surgió por querer realizar otras tareas mientras esperaba en una barra de progreso, por lo que se propuso el objetivo de crear un entorno de programación asíncrono, de forma que se puedan realizar varias tareas concurrentes sin que se bloquen entre sí.

Node.js está enfocado al desarrollo de servidores, especialmente a aquellos con una gran demanda de entradas y salidas de datos.

Actualmente su evolución está apoyada por la empresa Joyent y por la comunidad de programadores.

### 2.1.1. ¿Qué es Node.js?

Node.js es un entorno de programación pensado para realizar funciones de servidor. Permite la construcción de servidores de forma muy sencilla y rápida, además se puede aplicar para otros usos.

Originalmente, Node.js es un programa monohilo, lo cual quiere decir que solo utiliza un hilo en su ejecución, aunque con la librería "**cluster**" es posible convertirlo a multihilo.

Fue diseñado para crear programas que necesiten ser escalados, su único hilo es suficiente para dar servicio a un gran número de comunicaciones, aun así se pueden añadir más hilos y máquinas sin necesidad de realizar grandes cambios.

La ejecución es asíncrona. Esto significa que las funciones no son ejecutadas secuencialmente (si existen dos llamadas consecutivas, no es necesario que acabe la primera llamada para ejecutar la segunda).

Está pensado para ser un gestor de entradas y salidas. No ha sido diseñado para ejecutar gran cantidad de código, sino para realizar comunicaciones muy rápidas y abundantes, tanto con eventos locales como por comunicación por red.

Mantiene una arquitectura orientada a eventos, esto significa que la ejecución del programa viene determinado por los eventos, que son las entradas que recibe el sistema.

Utiliza la especificación de lenguaje ECMAScript para su programación, que es el mismo que utiliza JavaScript, que tradicionalmente se ha utilizado en el lado cliente.

La ejecución del código JavaScript es interpretada de la misma manera que en los navegadores, usando un motor basado en el V8 de Google. Node.js está programado en el lenguaje C++.

Para utilizarlo solo es necesario instalar el entorno de Node.js en la máquina. Los programas que ejecuta no necesitan ser compilados, simplemente se interpretan.

### 2.1.2. ¿Con que propósito ha sido concebido?

La especialidad de Node.js es gestionar Entradas y Salidas. Es decir, evitar que el cuello de botella del servidor sea el no poder aceptar más peticiones.

En el caso de un servidor web clásico, se dispone de un sistema multihilo y estos hilos atienden peticiones de la misma cola de clientes. Cuando un cliente es atendido por un hilo, este se bloquea hasta que el cliente ha sido atendido. Esto implica que si hay más peticiones concurrentes que hilos, la cola de clientes queda bloqueada hasta que las peticiones que se están atendiendo finalizan. En el caso de tener que realizar una operación de alto coste (como un acceso a memoria), tanto los clientes como el hilo están esperando a que finalice esa operación, por lo que este modelo desaprovecha una gran cantidad de tiempo en espera.

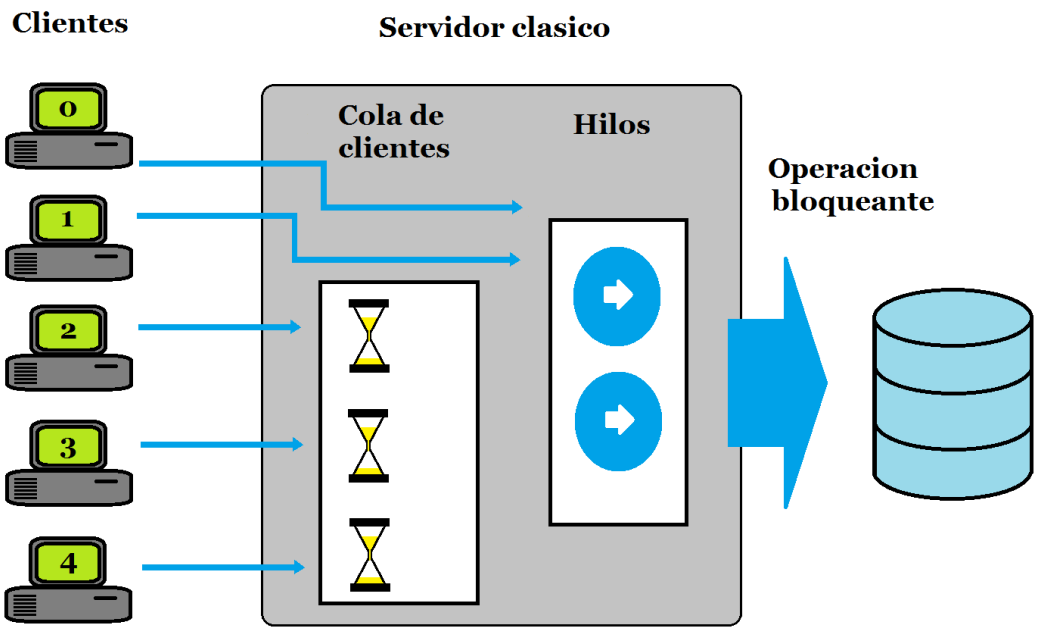


*Figura 1 - Modelo de servidor clásico*

En el caso de Node.js las peticiones realizan un recorrido distinto. Tratamos con un sistema monohilo, esto implica que solo podemos atender a un cliente a la vez, cuando un cliente es atendido, este solicita un recurso al hilo, el hilo delega la obtención de ese recurso al Sistema Operativo por una interfaz POSIX, y mientras el recurso está siendo obtenido, el cliente libera el hilo y se aparta a la sala de espera hasta ser llamado.

Cuando el hilo ha obtenido el recurso, el cliente vuelve a la cola de clientes, para que cuando llegue su turno, recibir el recurso solicitado.

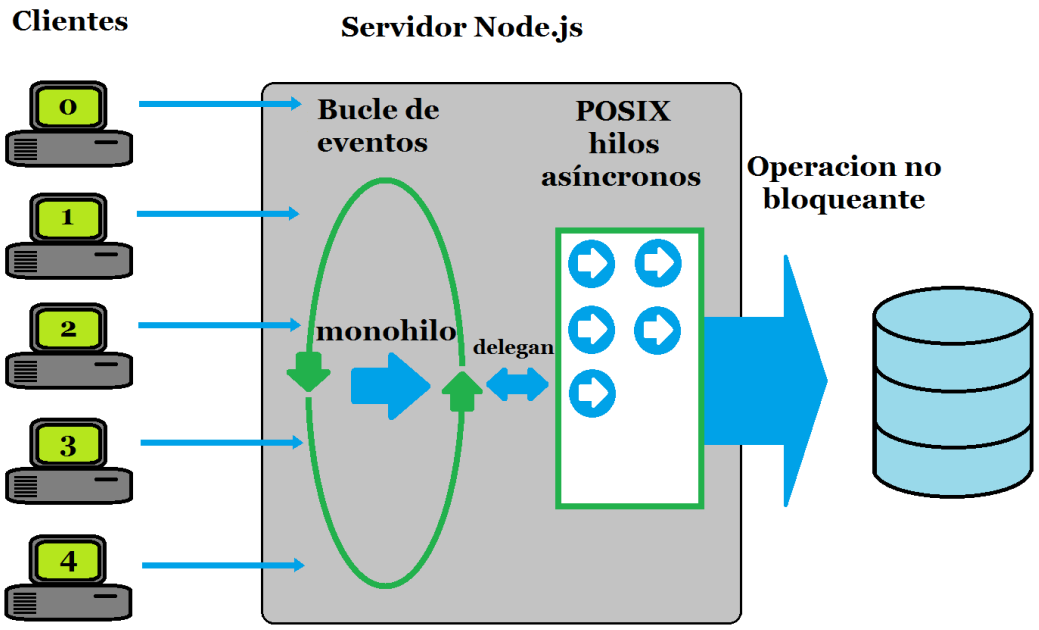


*Figura 2 - Modelo de servidor Node.js*

Es decir, los clientes en lugar de monopolizar un hilo hasta obtener el recurso, lo liberan permitiendo al resto de clientes avanzar en la cola.

Este sistema obliga a los clientes a pasar dos veces por la cola, una para solicitar un recurso y otra para recibirlo, pero esto no supone una pérdida de tiempo ya que la cola fluye constantemente.

### 2.1.3. ¿Qué es un sistema basado en eventos?

Mientras que en la programación secuencial es el programador el que determina el flujo del programa, en la programación basada en eventos el flujo del programa viene determinado por eventos.

**Los eventos**: Un evento es un suceso que cambia la ejecución del programa para ser atendido. Estos sucesos pueden ser: Interrupciones del sistema, entrada del teclado, comunicación con un puerto, etc.

**Los cambios en el programa:** Cuando sucede un evento, este cambia la ejecución del programa, que viene definido por el programador. Normalmente existe un bucle externo al programa, que registra estos eventos y ejecuta la función correspondiente. Node.js ya incorpora este bucle para registrar eventos de forma transparente al programador.

Esto no significa que cuando llega un evento, se abandona la ejecución anterior, simplemente el programa no ejecuta nada hasta que un evento sea activado. En el caso en que llamen dos eventos consecutivos, uno no anula o detiene al otro.

Para comprender este concepto podemos poner el ejemplo de un programa que recoge un dato y lo muestra. Tras arrancar el programa, este se queda en espera de algún evento definido en el bucle registrador de eventos. Cuando un usuario acciona el evento de recoger dato, el programa ejecuta el código necesario para esta tarea. Y cuando el usuario acciona el evento de mostrar dato, el programa lo muestra.

Con este ejemplo apreciamos que las acciones del programa escapan del control de programador, si un usuario acciona el evento de mostrar el dato sin haberlo leído puede surgir un error, por ello el programador debe de prevenir estas situaciones.

### 2.1.4. ¿Qué es un sistema asíncrono?

Cuando un sistema síncrono ejecuta una llamada, las instrucciones posteriores a esa llamada no se ejecutan hasta que esta ha sido completada.

Node.js es un sistema asíncrono. Esto significa que las llamadas y métodos son ejecutados de forma secuencial, pero sin esperar a que la anterior llamada haya finalizado.

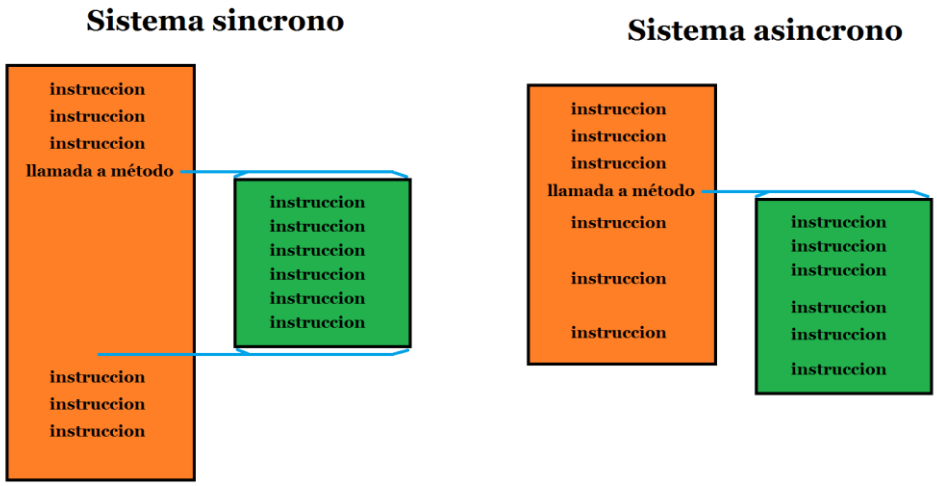


*Figura 3 - Funcionamiento de un sistema síncrono y asíncrono*

Un programa está sometido a esperas constantes como lectura de disco, llamadas a métodos de mucho coste, entradas de teclado, etc. Para que el programa se ejecute lo antes posible es necesario reducir esos tiempos de espera. Una solución consiste en continuar la ejecución del código sin esperar a la finalización de la llamada, a esto se le llama ejecución asíncrona.

Como podemos ver, con la programación asíncrona se reduce considerablemente el tiempo de ejecución, pero esto genera nuevos problemas, el más destacado es que las funciones no serán capaces de retornar valores. Este problema se puede resolver con

distintas técnicas, pero requiere cambiar el estilo de programación para adaptarse a este nuevo sistema.

### 2.1.5. Arquitectura

Node.js está organizado en una arquitectura por capas, que garantiza su estabilidad y retrocompatibilidad. Esta estructura consta de cinco capas, las cuales realizan la siguiente función:

**Dependencias:** Son librerías que contienen el motor V8, JavaScript, libuv, openssl, etc. Estas dependencias son las encargadas del funcionamiento básico.

**Interfaz binaria de la aplicación**: Permite la comunicación y proporciona acceso entre los módulos y aplicaciones con las dependencias.

**Biblioteca del núcleo:** Es la interfaz principal a través de la cual la mayoría de los módulos y aplicaciones de la capa superior de Node.js realizan operaciones de E/S, manipular datos, acceso a la red, etc. Algunos módulos y aplicaciones optan por enlazar directamente a la **Interfaz binaria de la aplicación** o incluso saltar directamente a las dependencias para realizar operaciones más avanzadas.

**Capa de abstracción binaria:** Consiste en una capa opcional utilizada para amortiguar los cambios entre la **Interfaz binaria de aplicaciones** y las **Dependencias** con las aplicaciones desarrolladas en la capa superior.

**Entorno de módulos y aplicaciones:** Son las aplicaciones desarrolladas en Node.js por los usuarios.

Esta estructura por capas garantiza la compatibilidad cuando se realizan cambios importantes entre versiones.

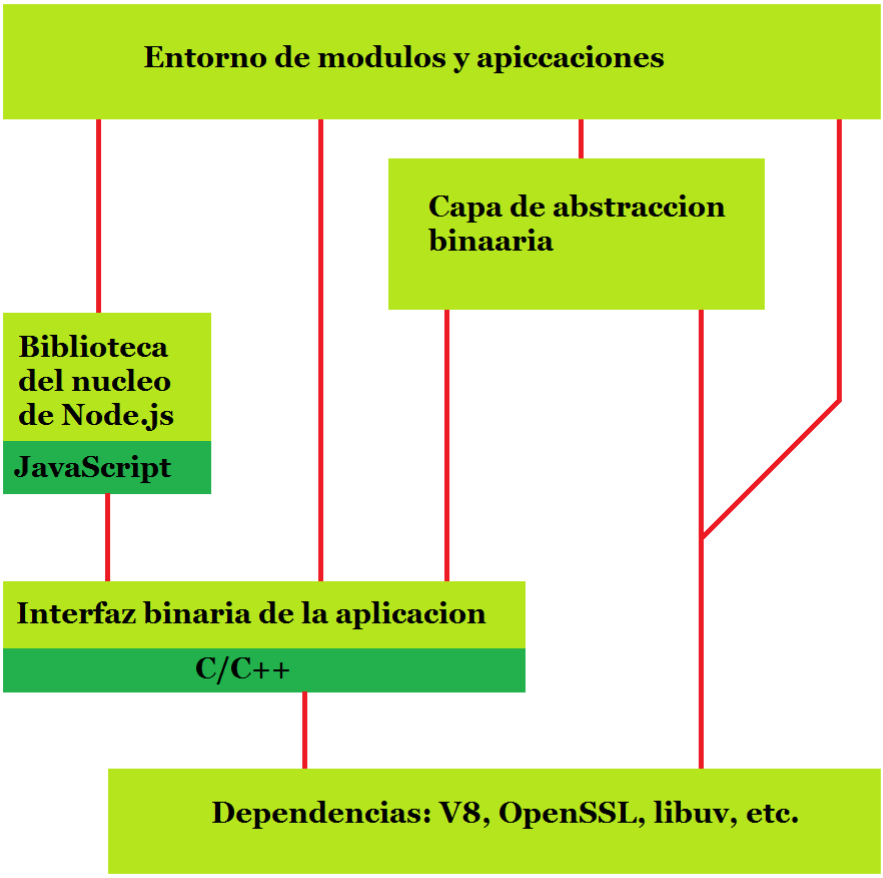


*Figura 4 - Arquitectura en capas de Node.js*

### 2.1.6. Política de desarrollo

Node.js crece y es mantenido por colaboradores individuales, los cuales desarrollan y proponen ampliaciones y mejoras del proyecto. Todas estas propuestas son evaluadas por el Comité de Dirección Técnica (TSC), compuesto por colaboradores clave que han demostrado experiencia técnica para el mantenimiento y la evolución del proyecto.

Una contribución a Node.js puede ser: correcciones de errores, mejoras de código, nuevas funciones, documentación, diseño, etc. Todas las aportaciones deben de ser retrocompatibles y no comprometer la estabilidad.

## 2.2. Módulos publicados

Igual que en el resto de lenguajes de programación, Node.js dispone de módulos para reducir la complejidad de los programas y simplificar el código principal, o

simplemente para realizar tareas que con solo JavaScript no se puede realizar. El entorno incorpora varios "módulos básicos" compilados en el propio binario, como por ejemplo el módulo de red, que proporciona una capa para programación de red asíncrona y otros módulos fundamentales, como por ejemplo Path, FileSystem, Buffer, Stream, etc.

Algunos módulos desarrollados por terceros los proporciona la página oficial de Node.js. Una fuente importante de estos módulos es GitHub, una página donde la comunidad de programadores puede publicar código abierto (https://github.com). Algunos pueden tener poco mantenimiento, por lo que su estabilidad y actualización puede ser variables.

Los módulos se pueden ser archivos ".node" precompilados, o archivos en JavaScript plano. Los módulos JavaScript se implementan siguiendo la especificación CommonJS, utilizando una variable de exportación para dar a estos scripts acceso a funciones y variables.

Pueden extender Node.js o añadir un nivel de abstracción, implementando varias utilidades middleware para utilizar en aplicaciones web, como por ejemplo los frameworks “connect” y “express”. Los módulos pueden instalarse como archivos simples o utilizando el Node Package Manager (npm), utilidad que facilita la instala, actualiza y resuelve dependencias de forma automática.

Estos módulos se instalan en un directorio por defecto (\nodejs\node_modules), si un programa necesita un módulo que no esté en ese directorio, se necesitará indicar una ruta en el programa para poder usarlos.

A continuación, se clasificarán algunos de los módulos más útiles de Node.js en dos apartados: Módulos incluidos y módulos comunitarios.

Los módulos incluidos son los que vienen con la instalación de Node.js y son creados por los mismos desarrolladores, por lo que se garantiza la estabilidad y la actualización. Los módulos comunitarios son los creados por terceros, si resultan ser de una relevante utilidad y estabilidad pueden pasar a incluirse en el código de Node.js.

### 2.2.1. Módulos incluidos

**http:** Puede ser el módulo más usado de Node.js, permite atender a las llamadas entrantes en cualquier puerto y devolverles una respuesta. Se usa principalmente para realizar funciones de servidor web pudiéndose aplicar también al resto de aplicaciones. La documentación se puede encontrar en la página oficial de Node.js.

**cluster:** Permite trabajar en multihilo, lo que ayuda a utilizar toda la potencia de la máquina. La documentación se encuentra en la página oficial de Node.js.

**dns:** Es un módulo especializado en resolver peticiones DNS, capaz de resolver la petición con la tabla DNS local o trasladar la petición a un servidor de mayor nivel.

**tls (ssl):** Permite establecer conexiones seguras usando OpenSSL utilizando el sistema de clave pública-privada. El servidor necesita un certificado para poder ser usado.

**OS:** Módulo que ofrece funciones de consulta sobre el estado del sistema operativo, como la versión del sistema, el estado de la memoria o el de la CPU.

### 2.2.2. Módulos comunitarios

**node-mysql:** Es un módulo de los más útiles, se puede encontrar en <https://github.com/felixge/node-mysql> y permite el acceso a una base de datos MySQL. La sintaxis para usar este módulo es muy similar a otros lenguajes (como PHP), y está acompañado de una buena documentación con instrucciones de uso, instalación y funcionamiento.

**node-formidable:** Del mismo autor de node-mysql, permite convertir ficheros en una estructura de datos (parsear), los archivos y la documentación están publicados en <https://github.com/felixge/node-formidable>. A igual que el modulo anterior, también tiene instrucciones de uso e instalación.

**AR-Drone:** Diseñado para el control de drones mediante un servidor, pueden enviar datos al servidor como imágenes de la cámara, coordenadas, etc y que este les devuelva instrucciones para dirigir el vuelo. Se puede aplicar usos como el reconocimiento facial o escaneo de grandes edificios. La página del módulo es <http://www.nodecopter.com/>, pero el módulo para descargar se encuentra en <https://github.com/felixge/node-ar-drone>, que es del mismo autor que los otros dos anteriores.

**node.js mongodb driver:** Mongodb ha publicado en su página oficial <http://mongodb.github.io/node-mongodb-native/2.0> un driver para poder usar Node.js con su base de datos. Está muy bien documentado y las instrucciones de instalación también vienen en la misma página, pero no se muestra el código. Generosamente también hace referencia al código abierto ofrecido por terceros, como **mongoose**, mostrando instrucciones de instalación y enlaces a la fuente.

**node-ffi**: Consiste en un complemento para poder llamar a funciones en C desde JavaScript. El código fuente se encuentra de nuevo en GitHub <https://github.com/node-ffi/node-ffi>. Se encarga de gestionar la traducción de tipos entre los lenguajes, pero se advierte que de no usarse correctamente, pueden surgir desbordamientos de pila.

**node-python**: Es una extensión para traducir código en python a JavaScript entendible para el V8, la fuente se encuentra en npm <https://www.npmjs.com/package/node-python> y es necesario el registro para la obtención del código.

**Passport:** Es una herramienta para poder conectarse con aplicaciones como Facebook Twitter, Google, etc. Tiene su propia página con documentación e

instrucciones de uso <http://passportjs.org/>, la página también contiene enlaces a programas y ejemplos en GitHub.

**Johnny Five:** Consiste en un módulo que ofrece una interfaz para la programación de microcontroladores, especialmente el Arduino. Tiene documentación y ejemplos en su página:<http://johnny-five.io/>.

**PDFKit:** Consiste en un módulo para la creación de documentos PDF mediante JavaScript, siendo construidos de forma similar a un documento HTML. La página con la información del módulo se encuentra en <http://pdfkit.org/>.

**Node Email Templates:** Permite el envío de correo electrónico. La documentación, ejemplos y descarga se encuentran en <https://github.com/niftylettuce/node-email-templates>.

**session:** Facilita la creación de sesiones de usuario y gestionar las cookies de sesión, el código lo podemos encontrar en <https://github.com/expressjs/session>.

# 3. Ventajas del uso de Node.js

Node.js permite la construcción de servidores y otras aplicaciones muy rápidamente empleando pocas líneas de código, la principal ventaja es un rendimiento constante sin ser notablemente afectado por el crecimiento de E/S. Otra ventaja es el posicionamiento Web, un servidor diseñado con esta tecnología tiene preferencia en los motores de búsqueda debido a que los algoritmos de búsqueda valoran positivamente el uso de tecnologías diferentes.

Las ventajas ofrecidas por Node.js se pueden clasificar en dos tipos:

**Cualidades intrínsecas:** Son propiedades beneficiosas independientemente de la aplicación implementada.

**Cualidades para la programación:** Son posibilidades que ofrece Node.js y JavaScript que mejoran y simplifican la programación de algunos tipos de aplicaciones.

## 3.1. Cualidades Intrínsecas

### 3.1.1. Gestión de entradas y salidas

Como ya se ha descrito, el diseño de Node.js está orientado a gestionar muy rápidamente las comunicaciones y la devolución de recursos, incluso es capaz de aceptar más llamadas que capacidad para atenderlas tiene la máquina. Esto significa que usando Node.js, la capa de comunicación de la aplicación deja de consumir recursos, y el crecimiento del volumen de comunicaciones no repercute en la máquina.

Debido a que el consumo de recursos de la capa comunicación no es significativo, un mayor porcentaje de la potencia de la maquina puede ser aprovechado por la lógica de la aplicación.

Estas propiedades lo hacen ideal para realizar funciones de FrontEnd, al ser un programa monohilo, no es apto para operaciones de gran coste computacional, para este tipo de tareas, Node.js solo debería hacer funciones de interfaz entre los clientes y el resto del sistema.

Node.js ofrece mucha escalabilidad, por ejemplo en un sistema en que los clientes necesiten peticiones que requieran mucho calculo, un FrontEnd con Node.js puede delegar a varias máquinas la lógica pesada de la aplicación.

### 3.1.2. Portable

Cualquier programa o Script en Node.js, no necesita ser compilado para trabajar en otras máquinas, al ser interpretado solo se necesita que la máquina destino tenga Node.js instalado.

Esta propiedad permite que cualquier programa pueda ser desarrollado sin tener en cuenta la arquitectura de la máquina. Si se necesitan módulos para su funcionamiento, estos pueden trasladarse directamente dentro del mismo directorio del programa.

### 3.1.3. Mismo lenguaje en el servidor que en el cliente, y facilidad para compartir datos

El lenguaje para la programación de Node.js es JavaScript. En un servidor web los clientes también usan JavaScript, de forma que ambas partes comparten el lenguaje, lo que facilita compartir información con los clientes, especialmente cuando se usa el formato JSON. Además, incluso es capaz de pasar funciones.

Es una ayuda para implementar el nuevo paradigma de comunicación cliente-servidor, donde se puede abandonar la comunicación petición-respuesta para entrar en una comunicación más compleja.

### 3.1.4. El propio Node.js puede generar la página

Un servidor web construido en Node.js puede generar páginas dinámicamente, esto es una buena idea si no se tiene muchos conocimientos de PHP.

Existen dos formas de generar una página íntegramente con JavaScript: Construir la página tratándola como un String, o mediante los métodos de inserción y modificación de bloques HTML. En ambas opciones, la creación de la página es muy abstracta y el programador no puede ver a simple vista el estado de la página, haciendo necesario interpretar el código JavaScript para ver el código HTML en claro.

Objetivamente, PHP es un lenguaje más eficiente para la generación de páginas web, es más maduro y con gran cantidad de código ya publicado. La integración del PHP con Node.js es posible, pero es bastante complejo.

### 3.1.5. Personalización del servidor

El desarrollo de un servidor en Node.js es muy sencillo y simple, permitiendo al programador centrarse en la funcionalidad de la aplicación.

Es importante poder elegir el nivel de complejidad de tu sistema. Con menos de 10 líneas puedes poner en marcha un sencillo servidor que devuelva una página HTML y aumentar la complejidad hasta ejecutar funciones, enviar datos al resto de clientes, incluso ejecutar comandos por consola.

## 3.2. Cualidades para la programación

### 3.2.1. Nuevo paradigma cliente-servidor

La comunicación en los servidores web clásicos consiste en recibir peticiones y devolver documentos, es decir, se basan en la petición-respuesta. Node.js cambia este modelo de comunicación permitiendo que el cliente se comunique con el servidor sin esperar respuesta o recibir datos sin la necesidad de refrescar o alterar la página. También permite la comunicación iniciada por parte del servidor al cliente, siempre y cuando el cliente este activo.

Como ejemplo de uso, para mantener actualizada una tabla en el cliente utilizando servidores clásicos, es necesario que el cliente refresque la página periódicamente para enviar la consulta al servidor y que este genere una nueva página con los nuevos datos. Mientras que con un servidor con el nuevo paradigma, el cliente solicita los datos al servidor mediante JavaScript y modifica tan solo la tabla de datos, el servidor solo recibiría la petición de datos y los enviaría, sin la necesidad de generar una nueva página, consumiendo menos recursos por cliente que con el modelo clásico.

El servidor también tiene la posibilidad de iniciar la comunicación, por ejemplo enviando advertencias a los clientes por inactividad.

### 3.2.2. Cambiar certificados SSL en caliente

En servidores clásicos, cuando se cambia el certificado SSL para ofrecer conexiones seguras, es necesario reiniciar el servidor, lo que origina una pérdida temporal de servicio si no se han tomado medidas. Un servidor basado en Node.js puede implementar un cambio de certificado en caliente que evite una pérdida de servicio.

### 3.2.3. Funciones de administración

Aunque no ha sido diseñado para eso, puede ser un lenguaje de scripting muy rápido y adecuado para la administración de máquinas. Ejemplos de este uso pueden ser: recogida de datos procesados, consulta del estado de la máquina, mantenimiento y apagado... En esta aplicación juega un papel importante la capacidad de Node.js para ejecutar comandos.

Es imprescindible que todas las máquinas a administrar dispongan de un servidor Node.js activo, lo que hace reconsiderar esta opción y volver a los script clásicos, que normalmente ya vienen instalados en sistemas Unix.

# 4. Tipos de Aplicaciones Recomendadas

Existen aplicaciones que por su naturaleza, se benefician de las características a las que puede dar soporte Node.js, a continuación se muestran algunos ejemplos de estas posibles aplicaciones. Aun así, sigue siendo el programador quien debe adaptar el programa a estas propiedades y saber aprovecharlas.

### 4.1.1. Chat

Un chat es una aplicación ideal para Node.js, es un programa donde una serie de clientes envían mensajes cortos y en gran cantidad hacia otros clientes mediante un servidor, el cual los encamina hacia sus respectivos clientes. Como ya hemos visto antes, la propiedad de Node.js de recibir y enviar mensajes encaja perfectamente en este sistema, los mensajes procesados son de corta extensión y no se hace necesario ningún sistema BackEnd que deba procesar esos datos, directamente se envían a los clientes correspondientes utilizando una lógica mínima.

Se puede añadir complejidad al sistema registrando las conversaciones en una base de datos, añadir videoconferencias, etc. En el caso de las videoconferencias, se manejan una mayor cantidad de datos entre clientes, pero el principio es el mismo, son datos que solo requieren reenvió y no necesitan ser procesados.

Demostrando la idoneidad de este tipo de aplicaciones encontramos un chat desarrollado por Nick Anastasov en <http://tutorialzine.com/2014/03/nodejs-private-webchat/> y el propio chat está activo en <http://tz-private-webchat.herokuapp.com/>. Tiene la desventaja de que solo admite una pareja de clientes por sala, pero el código es público para que cualquiera lo adapte a sus necesidades.

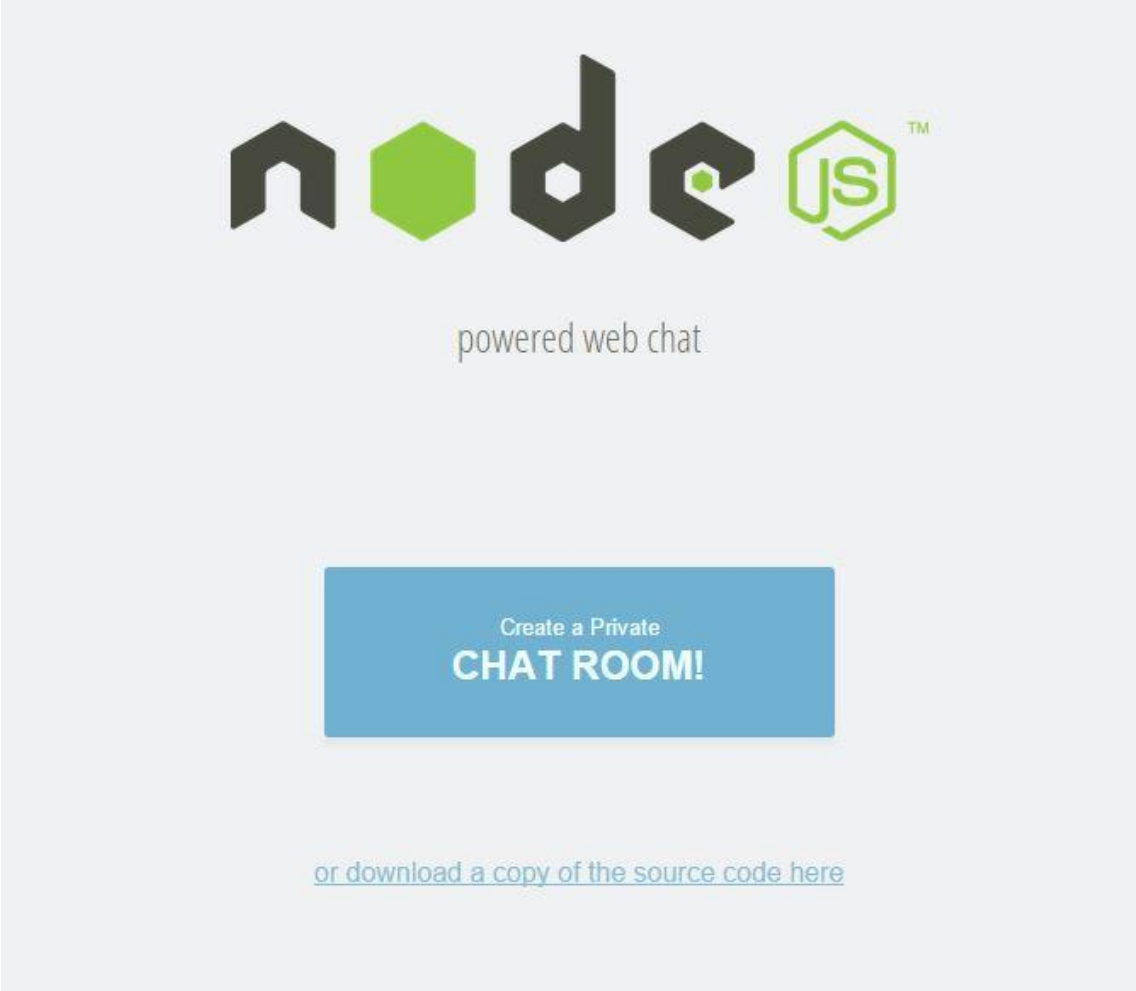


*Figura 5 - Pantalla principal de chat*

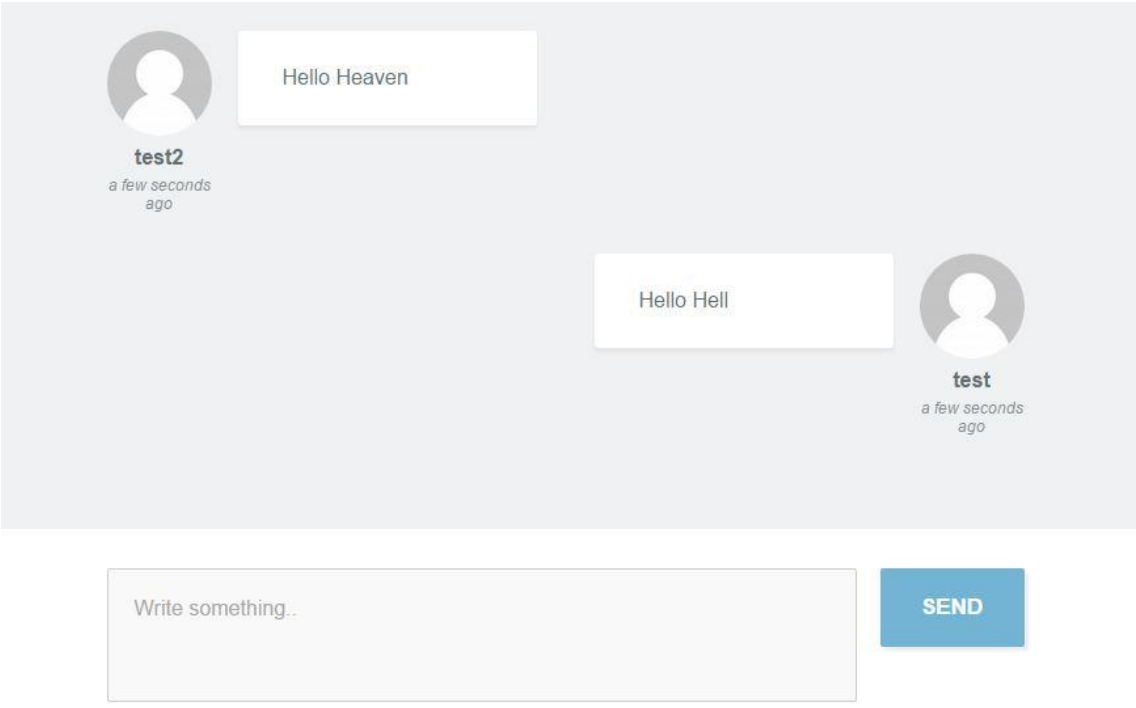


*Figura 6 - Prueba de conversación sobre el chat*

### 4.1.2. Datos en Streaming

La recepción y envío de datos en vivo se puede aplicar a videoconferencias, llamadas software, consulta de estado de equipos, etc. El modelo del programa es muy similar al chat con la diferencia de que el volumen de mensajes es mucho mayor.

Una aplicación de ejemplo podría ser **binaryjs**, el cual consiste en un módulo para el envío de datos en Streaming. La página principal se encuentra en <http://binaryjs.com/> y ponen como ejemplo funcional el envío de una imagen por websockets en este enlace <http://examples.binaryjs.com/hw.html>, también contiene enlaces al código fuente en GitHub.

Un segundo ejemplo es el envío de video desarrollado por Monteagudo, que es una simple aplicación que consiste en dos páginas, una emisora de video y otra receptora. El autor expone su aplicación en <http://www.jlmonteagudo.com/2012/10/emitir-video-con-tu-dispositivo-movil-con-node-js-express-js-y-socket-io/> y el enlace a la aplicación es <http://jlmonteagudo-cam.herokuapp.com/index.html>.

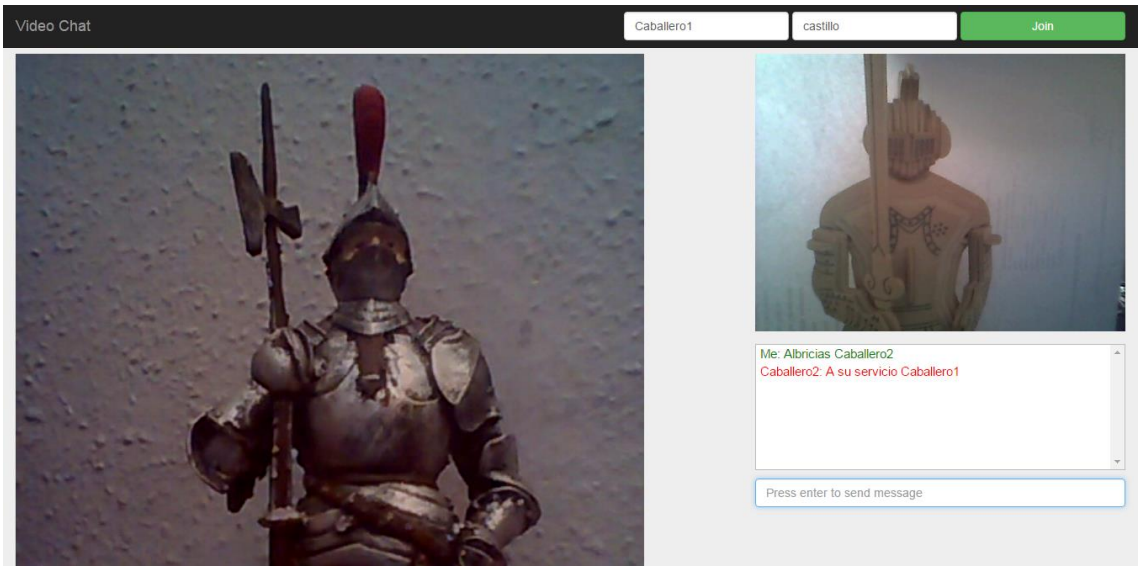


*Figura 7 - Captura realizada del videochat Peerchat*

### 4.1.3. Gestión de base de datos

El control de una base de datos es muy simple, un cliente envía al servidor que acciones quiere realizar sobre la base de datos, el servidor lo recibe, valida y actúa sobre la base de datos. Node.js realiza la función de servidor, donde puede recibir las peticiones de diversos clientes y actuar sobre la base de datos, y también puede realizar funciones de control de acceso, permitiendo solo operaciones a los clientes con autorización.

Como ya se ha explicado, desde que se envía una orden a la base de datos hasta que se recibe una respuesta, puede pasar un periodo de tiempo relativamente considerable.

Node.js no se queda esperando a recibir esa respuesta, sino que va realizando otras tareas y atendiendo a otras peticiones mientras se completa la acción anterior.

**HabitRPG** es un ejemplo de este uso, donde los clientes pueden registrar tareas conclusas y pendientes en una base de datos, además de validar a los clientes. La aplicación funciona en <https://habitrpg.com> y su código también es público en <https://github.com/HabitRPG/habitrpg>.

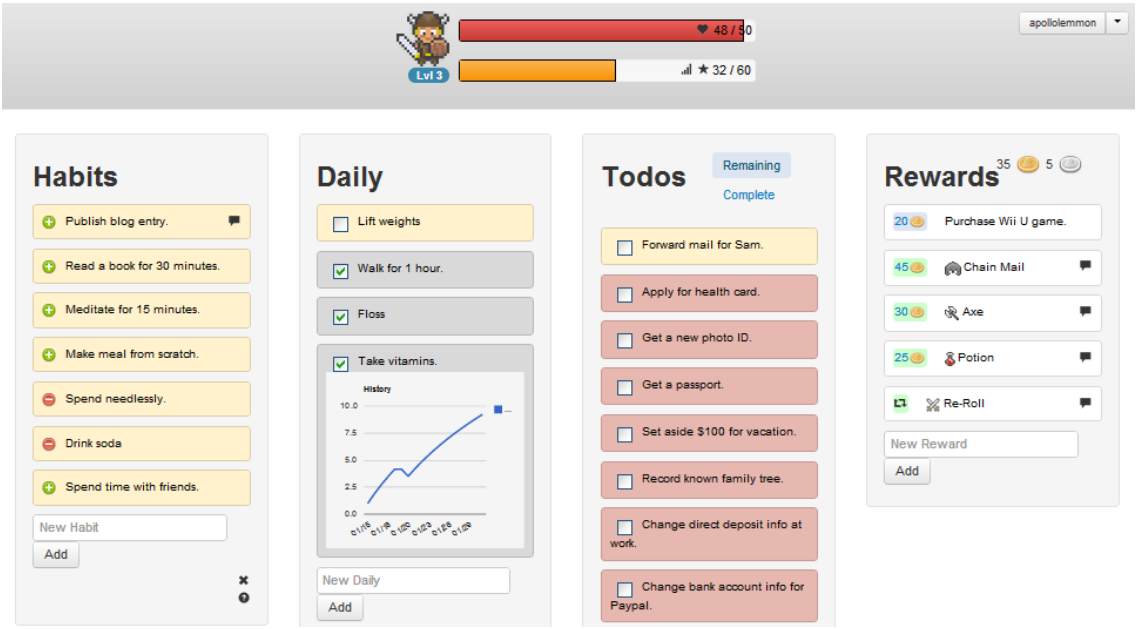


*Figura 8 - Pantalla principal de HabitRPG*

### 4.1.4. Servidor Proxy y balanceador de carga

Un balanceador de carga tiene la función de recibir peticiones y repartirlas entre varias máquinas, que individualmente, no serían capaces de aceptar esa carga.

Un servidor proxy, intercepta los mensajes de un cliente y los reenvía a la máquina destino suplantando la identidad del cliente. Es una función muy similar a la del balanceador de carga, pero esta requiere de mucha más lógica, pues se deben de recibir todos los paquetes e interpretar el mensaje del cliente para reenviarlo al destino. Esta tarea implica procesamiento y no es tan fluida como el balanceador de carga.

**node-http-proxy** es un programa que recoge ambas aplicaciones realizando funciones de Proxy como de balanceador de carga. El código se puede encontrar en <https://github.com/nodejitsu/node-http-proxy>, en este caso no se encuentran ejemplos de uso.

### 4.1.5. Servidor SMTP

Las funciones de un servidor STMP consisten en recibir mensajes, almacenarlos y enviarlos, siendo las operaciones de verificación muy cortas y los mensajes son normalmente abundantes y de poco tamaño, siendo una aplicación ideal para Node.js.

**Haraka** es un servidor SMTP desarrollado con esta tecnología. Está funcionando en distintas webs como: <https://www.emailitin.com/>, <http://www.fortantispam.com/>, <http://temp-mail.org/>, etc, su código es abierto y está disponible en<https://github.com/baudehlo/Haraka> y en <https://haraka.github.io/>.

### 4.1.6. Servidor DNS

Las funciones y las ventajas son muy similares a la gestión de una base de datos. Se reciben consultas para resolver direcciones de dominios y se consulta la información en la base de datos para devolverla al cliente.

**AtoniaDNS** <http://atomiadns.com/> es una empresa que ofrece servicios DNS, estos servicios están servidos usando Node.js. El código de sus servidores funciona bajo licencio ISC y publicado en <https://github.com/atomia/atomiadns>.

### 4.1.7. Juegos Online multijugador

Esta es una de las aplicaciones más abundantes, siendo en su mayoría demostraciones y ejemplos de las capacidades de Node.js. Son muy dispares en su diseño y no todas serian un buen ejemplo, pero por lo general, en este tipo de juegos, se está enviando un flujo constante de información entre gran cantidad de clientes y el servidor, convirtiéndolo, en principio, una aplicación ideal. En los casos en que la mayor parte del proceso se encuentre en el servidor, Node.js no podría asumir esa carga y debería ser usado como capa de comunicación.

**Word wide mace** es un juego experimental de **Google** donde un dispositivo móvil interactúa con un PC. El dispositivo móvil hace las funciones de mando para controlar el juego mientras que el PC ejerce de pantalla, ambos dispositivos se conectan por websockets. Los dos dispositivos cargan la página y el código JavaScript del mismo servidor web (la tecnología de este no es relevante), en cuanto el código está cargado, los clientes se comunican y se sincronizan con un servidor websocket construido con Node.js. El módulo Stage Builder mostrado en ejemplo gráfico de la aplicación es el encargado de construir el mapa de juego y no utiliza Node.js. La información de la aplicación se encuentra en <https://cloud.google.com/solutions/real-time-gaming-with-node-js-websocket> y la propia aplicación en <https://chrome.com/maze/>.

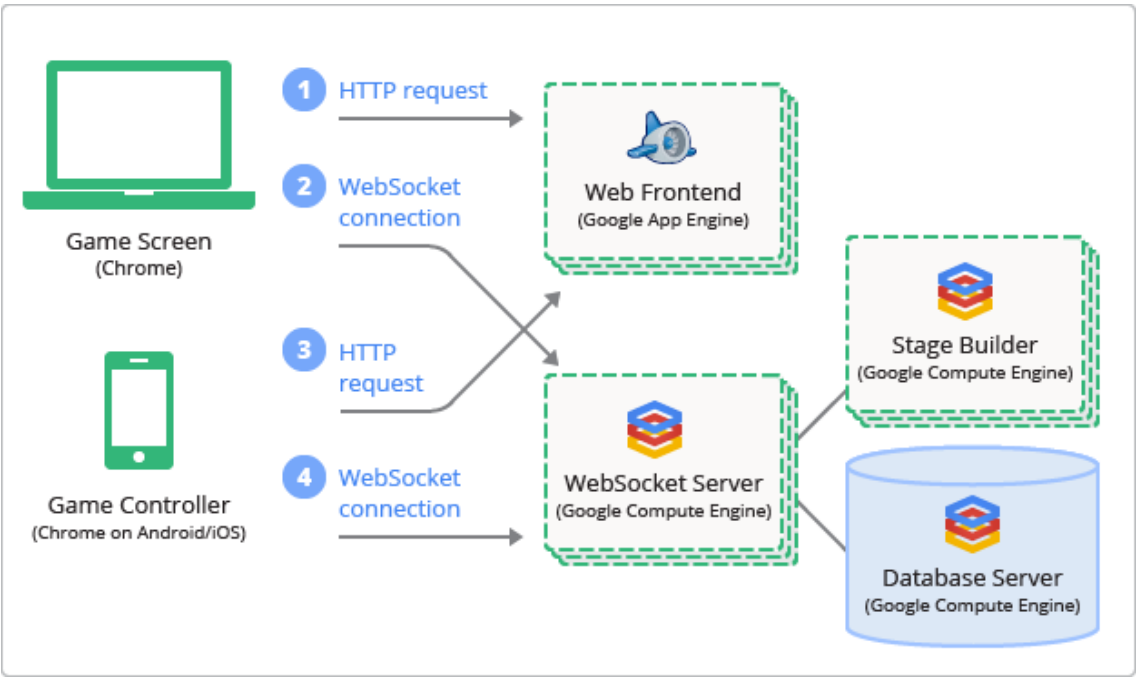


*Figura 9 - Esquema de diseño de Wold wide mace*

# 5. Inconvenientes del uso de Node.js

Node.js también presenta desventajas frente a otros frameworks. Algunos de los inconvenientes simplemente son aspectos a tener en cuenta a la hora de programar, otros son capacidades que añade Node.js pero que no pueden competir con otros entornos especialmente diseñados para esas tareas. También encontramos las que son funciones para las que Node.js no fue expresamente concebido, y por último, inconvenientes generados por la situación de Node.js en su entorno. Todos estos inconvenientes se han agrupado en tres categorías: **Inconvenientes técnicos**, **Inconvenientes de programación** e **Inconvenientes de entorno**.

Los **Inconvenientes técnicos** consisten en problemas que puede generar el entorno de Node.js debido a su modelo de trabajo, por ejemplo la utilización de un solo hilo.

Los **Inconvenientes de programación** suceden cuando se utiliza JavaScript para aplicaciones en las que no ha sido diseñado, por ejemplo no ha sido diseñado para crear paginas HTML sino para modificarlas. También surgen inconvenientes con la programación asíncrona, que requiere de diferentes técnicas de programación.

Los **Inconvenientes de entorno** son causados por la situación de Node.js en la sociedad y su demanda.

## 5.1. Inconvenientes técnicos

Son aspectos para los que Node.js no ha sido diseñado, por lo que las aplicaciones diseñadas en este entorno experimentarían problemas de rendimiento. Existen soluciones para algunos de estos problemas, las cuales se explican a continuación de haber sido enumerados.

### 5.1.1. Bloquear su único hilo

El proceso de Node.js se ejecuta en un único hilo, siendo un potencial riesgo su bloqueo. Una excepción o un bloque de código bloqueante, anularía todas las propiedades de gestión de entrada y salida que ofrece Node.js. El programador debe de prever todas las excepciones y tratarlas, igual que revisar el código potencialmente bloqueante.

Para resolver este problema, existe una librería llamada "**cluster**" que permite trabajar multihilo, si un hilo se bloquea, puede ser relanzado y los demás hilos siguen trabajando sin detener el servicio. Una técnica con esta librería es utilizar un hilo

maestro que detenga y relance los demás hilo en una planificación round-robin, si uno se bloquea será relanzado por el hilo maestro.

### 5.1.2. Solo gestiona Entrada y Salida

Node.js no hace que la lógica del servidor sea más rápida, lo que hace es aceptar peticiones y otras entradas muy rápidamente y delegar los procesos costosos al sistema operativo u otros medios. Puede darse el caso de que acepte más peticiones de las que el sistema puede atender, por lo que este puede sobrecargarse.

En el servidor que se necesite más capacidad de cómputo que de aceptación de peticiones, Node.js no es la mejor opción, pero aún podría usarse como capa de comunicación que distribuya la carga entre más máquinas.

### 5.1.3. Conocimientos de JavaScript

JavaScript es el lenguaje de Node.js, es muy conocido por los desarrolladores web, facilitándoles la adaptación al entorno de Node.js, pero la programación de servidores no suele ser trabajo de estos desarrolladores, en cambio un programador de servidores no puede tener conocimientos de este lenguaje, por lo que deberá de adaptarse. Esta situación dificulta encontrar desarrolladores que conozcan el lenguaje y necesiten la tecnología. En cualquier caso es necesaria una adaptación.

Una posible solución puede ser mantener un pequeño código en JavaScript y llamar a los métodos necesarios programados en otros lenguajes. Ya se han mencionado métodos para la llamada de funciones en Python, C o scripts por consola, pero sigue siendo necesaria la utilización de JavaScript.

## 5.2. Inconvenientes de programación

Son inconvenientes que requieren una mayor habilidad por parte del programador para solucionarlos. Algunos surgen por la novedad del nuevo paradigma asíncrono, que requiere una formación específica del programador, mientras que otros son problemas compartidos con otros lenguajes, pero con Node.js se acentúa aún más, como en la estructuración de código.

### 5.2.1. Estructurar código

Ya se ha visto que Node.js puede generar páginas web además de implementar las funciones del servidor. Esto obliga a tener una buena estructuración del código y separar muy bien cada aspecto. Si diseñamos nuestro sistema de forma monolítica, el

mantenimiento del servidor puede hacerse complejo, igual que la actualización y mantenimiento de la página.

Sería posible que los servicios de SMPT, HTTL y FTP se ejecutarán en el mismo programa, pero implica que un fallo en cualquiera de esas funciones bloquearía a las demás. Por lo que se recomienda que para evitar estos problemas se estructure bien el código en funciones y ficheros y un proceso distinto para cada servicio.

### 5.2.2. Generación de página

Anteriormente se ha enumerado la capacidad de generar una página web dinámicamente como una ventaja, pero más bien cuenta como una posibilidad, también se ha dicho que el lenguaje PHP está especialmente diseñado para la generación de las páginas web, mientras que Node.js se pensó para otros usos. Una forma de mitigar este problema sería integrar Node.js con PHP, ya existen algunas soluciones, pero todavía no son prácticas. Solucionar este problema sería un importante trabajo futuro.

Existen dos métodos para construir la página por JavaScript, pero ambos métodos pueden resultar complejos.

La primera opción consiste en generar la página como si fuera una variable de tipo string.

La segunda opción consiste en aprovechar la propiedad de JavaScript para generar bloques de HTML, propiedad que PHP no cumple, ya que solo inserta código sin conocer la sintaxis.

Este último método tiene la ventaja de que enviando el código JavaScript al cliente este puede autogenerarse la página, restando carga al servidor. En cualquier caso, existe el inconveniente de que al programador le resulte más difícil interpretar el código final HTML, en lugar de tener un fichero con la página y el código para generarla.

### 5.2.3. Programación asíncrona

Ya se mencionó que una de las ventajas de Node.js, es su programación asíncrona, pero puede representar un reto para el programador que no comprenda su funcionamiento. En los programas asíncronos, cuando el código efectúa una llamada a un método, continúa la ejecución principal sin esperar a la finalización del método llamado.

Tanto el método llamado como el llamante prosiguen la ejecución, por lo que si el método llamado devuelve algún valor mediante la instrucción “return”, el método llamante se encontrará ya avanzado y no recibirá el valor del método llamado.

Esta característica, fuerza a organizar el programa de una forma diferente al de la programación secuencial, y a tomar medidas para obtener resultados de los métodos

hijos a los métodos padre. Existen soluciones como esperar a la finalización del método cuando se necesite su resultado o en consultar el estado del método de forma activa, estas y otras soluciones se encuentran en <http://elmanantialdebits.blogspot.com.es/2013/08/que-es-eso-de-la-programacion-asincrona.html> y se enumeran a continuación:

**Pipes:** Consiste en conectar la operación con otra antes de solicitar su realización.

**Interrupción:** El programa tiene una interrupción, que cuando es llamada por el método en su finalización, recupera su valor resultado. Es importante ver que este código está desligado del punto en el que se solicitó la operación mediante la llamada asíncrona.

**Polling:** Se basa en preguntar cada cierto tiempo si se ha completado la operación. No es una solución eficiente ya que se pierde tiempo en consultar el estado del método.

**Eventos/mensajes:** Es una mejora sobre polling. Cuando termina la operación se envía un mensaje que se encola. El programa no consulta si una operación en concreto se ha completado, en cambio sólo comprueba si ha llegado algún mensaje. Esto le indica que una operación de las muchas en curso se ha completado. Cada mensaje tiene un indicador de la operación completada y deberá ser respondido correspondientemente, de forma similar a como se hace con las interrupciones.

**Hebras o fibras:** Se usan llamadas síncronas, pero en lugar de que bloqueen el programa, se dan opciones de por dónde se puede continuar la ejecución. En el caso de las hebras, es el sistema operativo el que decide cómo planificar la ejecución, mientras que en el caso de la fibra es el propio programa el que lo hace.

**Sincronizar:** Es la solución más sencilla. El programa se sigue ejecutando de forma asíncrona hasta que sea necesario la respuesta de la llamada, entonces se espera a que la llamada acabe. Esta opción mantiene un programa asíncrono todo lo posible, pero pueden darse casos en que todo el código sea dependiente de sus llamadas, este caso anularía completamente las propiedades asíncronas de Node.js Por lo que es la solución menos recomendada.

**Retrollamadas:** Cuando se realiza una llamada se indica que llamada debe de ejecutar en cuanto el método acaba. Esto hace que la llamada no vuelva al método padre, sino que bifurca la ejecución.

Por último, la depuración y detección de errores es compleja, como los métodos no se finalizan secuencialmente, es difícil conocer en qué punto puede fallar un programa mediante mensajes indicando el estado del programa.

En contrapartida, un programador acostumbrado a la programación paralela tendría mayor destreza en este tipo de programación, debido a la similitud de tener varias tareas simultáneas con dependencias entre ellas.

### 5.2.4. Estabilidad de módulos

En Node.js los módulos son elementos importantes, permitiendo funcionalidades y ofreciendo código ya realizado y probado. Estos módulos tienen la ventaja de que varios usuarios pueden haber publicado uno, solucionando un mismo problema de formas muy distintas y ofreciendo diversas alternativas a los programadores. En cambio, esto también puede suponer un inconveniente por la labor de analizar y comprender su funcionamiento de estos antes de su uso. Sería mucho más sencillo para el programador que Node.js incorporara estas funcionalidades, permitiendo una mayor portabilidad de código y sencillez en la programación.

Con el tiempo, algunos de estos módulos pueden ser incluidos en el núcleo de Node.js, funcionalidades básicas como los inicios de sesión o el acceso a base de datos.

## 5.3. Inconvenientes de entorno

Aquí se agrupan inconvenientes sociales, de situación, de demanda, etc. Todos los aspectos que afectan negativamente sin ser problema de Node.js o de su programación.

### 5.3.1. Demasiado nuevo

Node.js es un lenguaje joven, y aunque existe una gran comunidad de desarrolladores, todavía no hay tanto código publicado como en otros lenguajes, dificultando en aprendizaje y resolución de problemas. Este es un problema que rápidamente se está solucionando, pero la principal fuente para aprender un lenguaje de programación son los tutoriales y guías en internet, los cuales todavía son escasos. Existen muchos tutoriales sobre la utilización de JavaScript, pero los más necesarios y escasos en el momento de redactar este proyecto son las que explican el funcionamiento y técnicas de programación asíncrona, que sería un gran trabajo para el futuro.

### 5.3.2. Inseguro

Al ser un lenguaje interpretado, se hace más sencilla la inyección de código. Las variables en JavaScript, pueden ser también funciones, ya que cualquier variable recibida por el cliente debe de pasar por un robusto método de validación antes de poder ser usada.

A esto se une que al llevar poco tiempo en uso pueden existir bugs y agujeros de seguridad no descubiertos. Otros servidores convencionales como Apache, han sido depurados y probados en multitud de casos, creando una tendencia a usar estos sistemas ya probados.

Con Node.js, cada nuevo programa es un nuevo servidor, probar cada nuevo servidor creado es un gran trabajo, y conseguir el nivel de testeo de un servidor ya implementado con varios años de uso es muy complicado.

### 5.3.3. Hosting

La mayoría de los clientes de los servicios de hosting necesitan publicar páginas web, estas páginas normalmente son diseñadas en PHP. Este hecho dificulta el uso de Node.js en servicios web ya en que su mercado abunda la demanda de servidores clásicos.

A esto se une que un administrador de sistemas que ofrezca servicios de hosting de Node.js, le será más complicada la comprobación del código de sus clientes y controlar su funcionamiento, incluido la detección de fallos de seguridad, especialmente con la capacidad de ejecutar comandos. Es decir, el servidor escapa al control del administrador del host.

Si un cliente necesita publicar una aplicación con Node.js necesita un host dedicado para sus necesidades.

# 6. Tipos de Aplicaciones Problemáticas

Igual que existen aplicaciones que aprovechan las propiedades de Node.js, existen otras con las que esta tecnología es ineficaz para sus funciones más destacables. La siguiente lista es orientativa, en la que se enumeran las aplicaciones generalmente contraproducentes.

Al ser aplicaciones no recomendadas para ser implementadas en Node.js, existen pocos ejemplos de su uso.

**Servidores HTTP:** Anteriormente se ha explicado que la publicación de páginas web con Node.js genera algunos problemas, como puede ser la creación o migración de páginas entre servidores. Un servidor funcionando en ese entorno, es muy rápido con la generación de páginas, la complejidad se sitúa en la capacidad del programador para ver el código claro.

Cuando se diseña una página web, los diseñadores no tienen en cuenta la creación del servidor, sino que delegan este problema a servidores ya publicados, no haciéndose necesario programar un servidor existiendo ya unos que funcionan.

Para cambiar una página web de host con Node,js, sería necesario mover el servidor entero o adaptar el programa del servidor origen y el servidor destino. Si el cambio se efectúa desde un servidor clásico a uno con el entorno o viceversa, podría ser necesario reescribir la página entera adaptándolo al servidor destino.

Un ejemplo de un servidor web en Node.js es **Open Marriage**, donde se publica una página web estática, el código de la página web está integrado en el código del servidor. La dirección código fuente del servidor es <https://github.com/ericf/open-marriage>.

**Tratamiento, conversión y cómputo:** Se ha dicho que Node.js es muy veloz en la gestión de comunicaciones, no obstante para el cálculo pesado no ha sido diseñado, por lo que programas de conversión de ficheros online u otros tratamientos de datos, tendrían una repercusión negativa sobre su funcionamiento. Una alternativa para solucionar este problema, sería dejar a Node.js solo como FrontEnd, delegando esos cálculos a un programa especialmente diseñado para esa tarea.

Ya se ha comentado que Node.js trabaja con un solo hilo y sobrecargar ese hilo implica que dejará de atender a la cola de clientes, interrumpiendo así su principal propiedad de atender llamadas.

No existen aplicaciones de este tipo implementadas en el entorno, por lo que no es posible mostrar ejemplos.

**Juegos Online basados en procesamiento del servidor:** Este tipo de aplicaciones suele necesitar que el servidor realice procesamiento con los datos recibidos de los clientes, como en las aplicaciones anteriores. La implementación

correcta de este sistema sería que Node.js realice funciones de FrontEnd mientras que la lógica de negocio y el resto de procesos se implemente utilizando otra tecnología.

**Tiendas y operaciones bancarias Online:** Suelen ser aplicaciones en las que se procesan datos críticos y privados. Para la construcción de estas aplicaciones se suele requerir de módulos para determinadas funciones como el acceso a la base de datos y el inicio de sesión. Se ha explicado que estos módulos son desarrollados por terceros y existen gran cantidad de ellos para el mismo objetivo, por lo que un desarrollador necesita probar repetidamente los módulos que se desean usar para la aplicación y verificar que los comportamiento son legítimos y seguros para el comercio (como fallos de seguridad, no guardar registros fuera de servidor, etc).

# 7. Análisis de aplicaciones en uso

## 7.1. Aplicaciones destacadas

Existen gran cantidad de aplicaciones ya implementadas con Node.js. El análisis de un subconjunto representativo de estas aplicaciones nos permitirá contrastar las propiedades teóricas antes mencionadas con casos reales. Algunas de las aplicaciones tan solo son ejemplos funcionales para demostrar las capacidades del entorno mientras que otras son comerciales y de uso extendido.

Las aplicaciones escogidas para analizar se presentaran ordenadas por la cantidad de usuarios que las utilicen. En cada aplicación se clasificara su tipo, se analizará su funcionamiento y se ofrecerán enlaces al código y la propia aplicación.

### 7.1.1. LinkedIn

Aplicación: **LinkedIn.**

Descripción: Red social orientada al entorno laboral, ofreciendo la posibilidad de encontrar trabajo y recomendaciones de otros usuarios.

Dirección Web: <https://es.linkedin.com/>.

Tipo de aplicación: Red social.

Arquitectura: Anteriormente, LinkedIn funcionaba con 30 servidores, los cuales tenían implementados Ruby on Rails como ForntEnd y otros programas en BackEnd. Tras la decisión de migrar el código de Rubi on rails a Node.js, se redujo la carga de los servidores significativamente. El BackEnd también fue fusionado al FrontEnd de Node.js.

Unas fuentes dicen que se redujeron de 30 a 3 servidores, otras fuentes indican que la reducción fue de 16 a 4, pero todas coinciden en que el cambio fue beneficioso y que redujo el hardware necesario.

Características: Las fuentes también indican que este cambio aprovecha mejor las habilidades de sus programadores en JavaScript, también se mejoró el rendimiento de las máquinas, reduciendo la carga de la memoria y funcionando hasta 20 veces más rápido.

El código fuente de la página no está publicado, aun así se puede deducir el funcionamiento, consistiendo en una base de datos que albergue la información de los usuarios y unos algoritmos calculan el valor que tiene cada usuario. Cuando un cliente

se conecta, este debe de recibir su información de la base de datos y los algoritmos le indican que usuarios pueden ser de su interés.

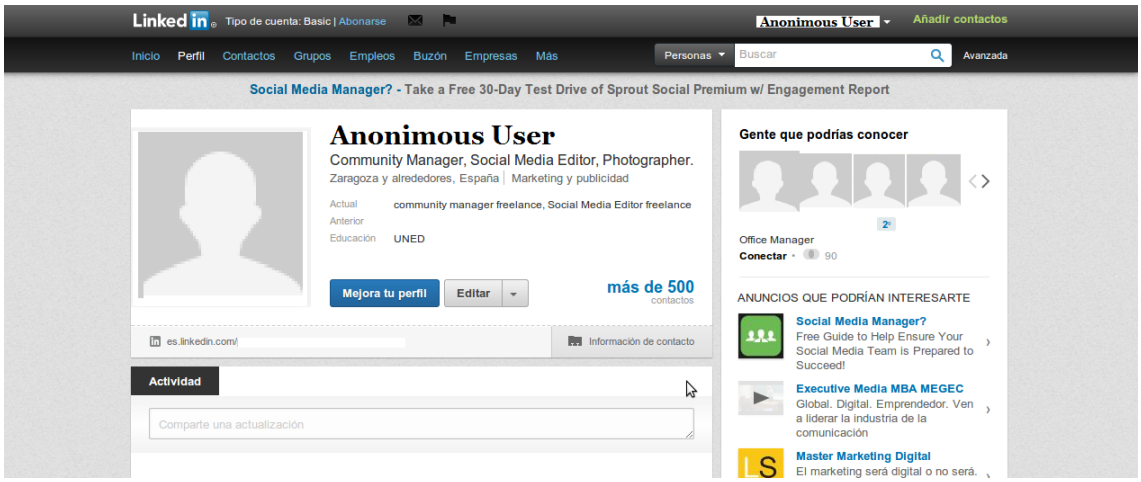


*Figura 10 - Perfil de Linkedin*

### 7.1.2. BrowserQuest

Aplicación: **BrowserQuest.**

Descripción: Juego online de navegador, todos los jugadores comparten un mapa y ven la posición de los demás jugadores.

Dirección Web: <https://github.com/mozilla/BrowserQuest>.

Dirección del código: <https://github.com/mozilla/BrowserQuest>.

Tipo de aplicación: Juego multijugador MORPG (Multiplayer Online Real Progres Game).

Arquitectura: Según su arquitectura, el servidor envía el código y los datos que necesita el cliente como funciones, mapas, imágenes... El servidor también recibe la información de las acciones de cada cliente y las difunde entre los demás clientes.

El cliente debe de almacenar la información de su personaje como equipo y logros. Cuando el cliente realiza un cambio se comunica al servidor y el servidor difunde las posiciones y sus equipos (inventarios) entre los demás clientes.

Características: Este modelo es muy eficiente, pero inseguro, el cliente puede cambiar los datos de su personaje sin ser comprobados por el servidor, como ponerse el mejor equipo al principio del juego. Para una aplicación donde el cliente está limitado por su progreso, este modelo no serviría y debería ser el servidor quien almacene los datos de los personajes.

Por otro lado, si un usuario desea cambiar de maquina cliente perdería el progreso del juego. El progreso se almacena en cookies y esta sería la única manera de transportar los datos del cliente.

Este es un ejemplo adecuado del uso de Node.js, el servidor solo reparte datos de poco peso entre varios clientes. No se usa base de datos, ni operaciones de mucho peso ni bloqueantes. El código fuente esta publicado en GitHub para todo aquel que desea aprender más sobre Node.js.

El servidor íntegramente esta hecho en Node.js y el juego es una demostración de sus capacidades. Se aplica el nuevo paradigma antes descrito en el que no existe la comunicación petición-respuesta, sino que hay una constante comunicación de datos entre los clientes y el servidor manteniendo la página sin ser recargada.


*Figura 11 - Gráficos de Browserquest*

### 7.1.3. HabitRPG

Aplicación: **HabitRPG.**

Descripción: Se trata de un gestor de tareas y de hábitos imitando un juego RPG, está basado en el juego antes descrito, **BrowserrQuest**. El usuario apunta sus tareas, buenos y malos hábitos... con el objetivo de que su personaje progrese con las buenas acciones y se perjudique con las malas acciones que el usuario realiza, motivando a este a coger buenos hábitos.

Dirección Web: <https://habitrpg.com/>.

Dirección del código: <https://github.com/HabitRPG/habitrpg>.

Tipo de aplicación: Gestor de tareas, registro de eventos en base de datos.

Arquitectura: Básicamente consiste en una interfaz que registra datos en el servidor, el cliente solo puede insertar datos y consultarlos sin eliminar datos anteriores.

El cliente solo lee las acciones registradas en el servidor y escribe acciones nuevas, no se necesita ningún tipo de procesamiento por ninguno de los dos lados, manteniéndose el paradigma de comunicación de datos sin recargar la página.

Características: El modelo utilizado es muy similar al de **BrowserQuest**, con la diferencia de que en lugar de almacenarse la información en el cliente, se almacena en una base de datos en el servidor. Tampoco existe un mapa y no se difunden los datos de un cliente hacia el resto, sino que consiste principalmente en recoger información en la base de datos.

Es un programa muy simple, que se basa en recoger datos y devolverlos cuando se piden. Pero con el uso de Node.js obtienes mayores beneficios, ya que puedes atender a muchos clientes simultáneos que piden pocos recursos.

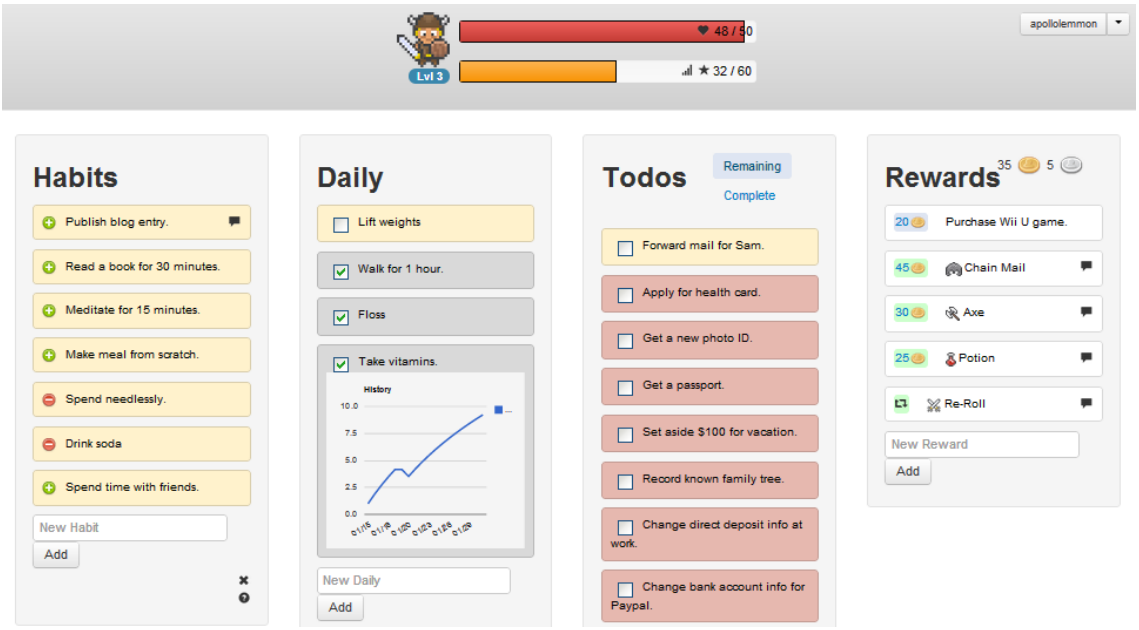


*Figura 12 - Pantalla principal de HabitRPG*

### 7.1.4. ChessHub

Aplicación: **ChessHub.**

Descripción: Es un servidor para jugar al ajedrez online. Permite crear y jugar partidas, ver partidas en curso y registrar jugadores.

Dirección Web: <http://chesshub-benas.rhcloud.com/>.

Dirección del código: <https://github.com/benas/gamehub.io>.

Tipo de aplicación: Juego Online dos jugadores.

Arquitectura: Este es otro buen ejemplo del uso correcto de Node.js. Como indica la figura 1, los clientes envían los movimientos al servidor para reenviarlos al resto de clientes que los necesite. Estos movimientos quedan registrados hasta el final de la partida.

Es un servidor muy sencillo, los datos enviados por un cliente son devueltos al resto de clientes, y estos son tan pequeños como una cadena de no más de 10 caracteres.

Los clientes deben permanecer a la escucha, cuando reciben un movimiento, instantáneamente lo muestran en el tablero, y si el cliente es un jugador, puede enviar un movimiento al servidor.

Características: La aplicación mantiene un cierto nivel de seguridad, un jugador no pueden enviar movimientos en cualquier momento, por lo que el servidor y los clientes validan los movimientos enviados y los recibidos. Los clientes jugadores disponen de un semáforo para enviar estos movimientos.

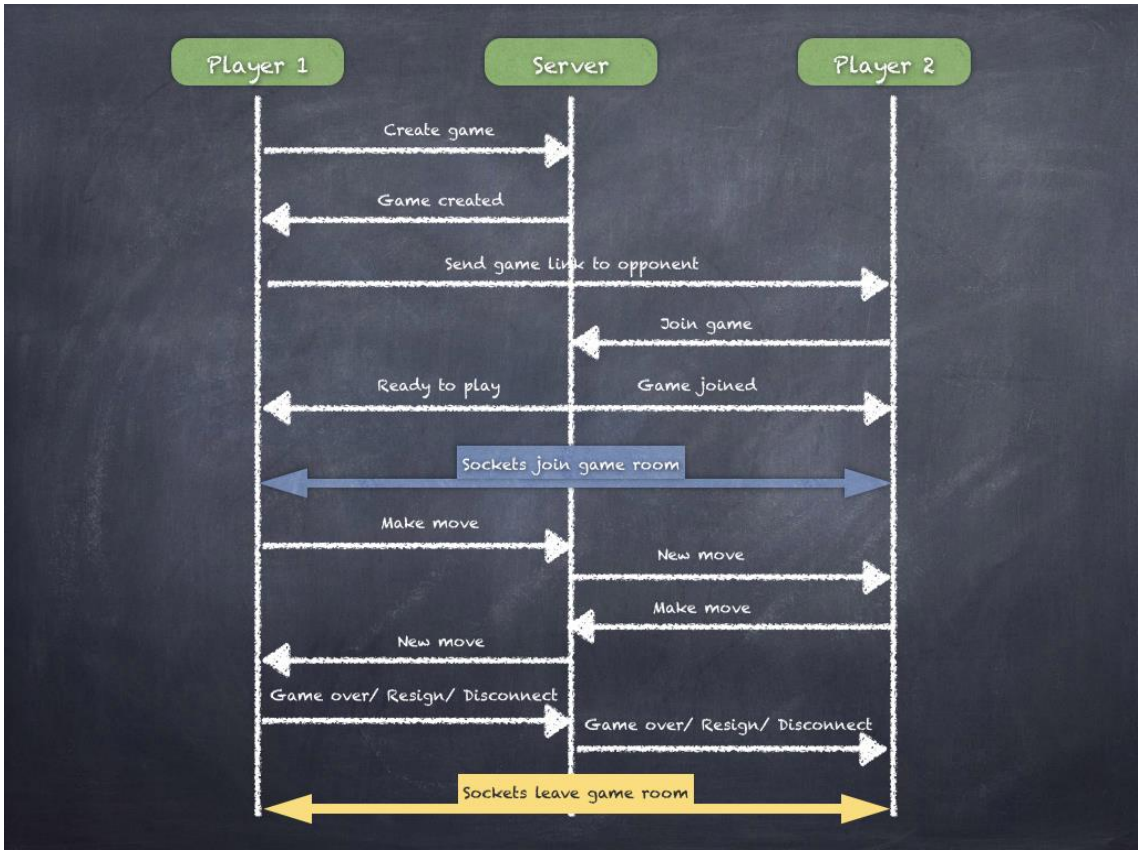


*Figura 13 - Esquema de las comunicaciones en Chess Hub*

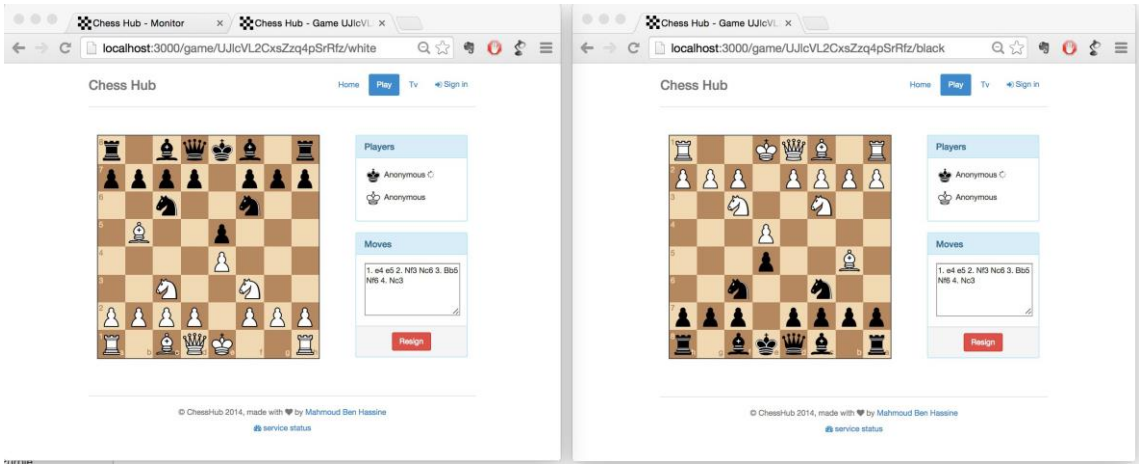


*Figura 14 - Una partida de ChessHub*

### 7.1.5. Open marriage

Aplicación: **Open marriage.**

Descripción: Pagina Web para informar y registrar invitados a una boda.

Dirección Web: <http://leslie-eric.us/>.

Dirección del código: <https://github.com/ericf/open-marriage>.

Tipo de aplicación: Pagina Web.

Arquitectura: Consiste en una simple página estática donde te registras en la base de datos como invitado, la única comunicación de cliente a servidor es para registrarse.

Características: Usar Node.js es bueno, pero existe muy poca demanda de comunicación por parte del cliente, que no se aprovecha la capacidad del entorno, en el manejo de entradas y salidas. Por otro lado, es una aplicación tan sencilla que no compensaba construir un servidor especialmente para ella, pero el autor se felicita por haberlo programado y se anima a otros a que experimenten también con el entorno, igualmente, ya he mencionado que la construcción de un servidor HTTP en Node.js es una tarea muy sencilla, y más cuando se sirven unas pocas páginas estáticas.

## 7.2. Más referencias de aplicaciones

Se pueden encontrar más ejemplos de aplicaciones usando Node.js en los siguientes enlaces:

**Nodejitsu**

Descripción: Breve lista de aplicaciones usando Node.js.

Dirección: <http://blog.nodejitsu.com/ten-open-source-node-js-apps/>.

**Reddit**

Descripción: Foro de programación, el tema enlazado consiste en la búsqueda de aplicaciones que unen Node.js.

Dirección: <http://www.reddit.com/r/node/comments/2nxnab/does_anyone_know_of_a_realrealw_complex/>.

**GitHub**

Descripción: Cosiste en una extensa lista de empresas y programas, los cuales utilizan Node.js.

Dirección: <https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node>.

# 8. Un Marco Conceptual para evaluar la idoneidad del uso de Node.js

Cuando se desea construir una aplicación, se prevé que esta deba dar soporte a una serie de características y requisitos funcionales y no funcionales. Una vez se ha analizado la aplicación, se debe analizar si los frameworks y lenguajes de desarrollo permiten implementar esos requisitos de forma correcta, eficiente y productiva. En nuestro caso, se comparan con las ofrecidas por Node.js.

## 8.1. Definición del marco

En el desarrollo de una aplicación, es durante la fase de planificación donde se elige el entorno de desarrollo, para ello es necesario realizar un análisis de requisitos y reunir todas las propiedades que debe tener el sistema para poder elegir posteriormente la tecnología con la que desarrollar la aplicación. Es conveniente realizar un listado de estas tecnologías disponibles con sus propiedades y desventajas, realizar una comparación y escoger la más adecuada para el sistema.

Si esperamos que la aplicación a desarrollar reciba un volumen bajo de peticiones por parte de clientes o pocas comunicaciones, Node.js no sería la opción más adecuada. Más adelante se estudia que Node.js comienza a ser realmente efectivo en peticiones con una carga de comunicación abundante. Esto implica que para las pequeñas aplicaciones podría usarse tecnología clásica, mientras que para aplicaciones que acojan a muchos clientes o que mantengan una transferencia de datos constantes, Node.js es la opción ideal.

La figura 15 contiene una relación de tareas y características con una recomendación de si es apto o no para ser desarrollado en Node.js.

| Tabla de idoneidad de Node.js | |
|---|---|
| Características | Recomendación |
| Aplicaciones basadas en procesar mucha lógica. | no |
| Necesidad de gestionar gran cantidad de comunicación. | si |
| Compatibilidad tecnológica para el diseño web (CMS, PHP,etc) | no |
| Máquinas poco potentes | si |
| Aplicaciones sensibles y críticas, que necesitan seguridad. | no |
| Aplicaciones con interacción entre clientes y servidor. | si |

*Figura 15 - Tabla de decisión de la idoneidad de Node.js*

## 8.2. Toma de decisiones

A continuación se analizaran los requisitos de distintos programas para tomar la decisión de si el uso de Node.js es apropiado para determinada aplicación.

### 8.2.1. Clara.io

Comenzamos analizando **Clara.io** <https://clara.io>, consiste en un programa de diseño 3D online a través del navegador. El servidor está dividido en un FrontEnd y en un BackEnd, las operaciones de renderizado 3D que debe enviar cada cliente son considerablemente pesadas y las comunicaciones entre cada cliente y el servidor son abundantes y constantes.

Los requisitos importantes son:

**Transferencia de datos:** La aplicación se ejecuta en su totalidad en el servidor, por lo que el cliente tan solo ha de enviar y recibir los datos de la interfaz. Estos datos son bastante abundantes por cada cliente, debido a que en tiempo real el cliente puede mover la figura que está siendo visualizada y recibir del servidor la imagen de la figura desde el nuevo ángulo, la velocidad de refresco de la imagen y su calidad es considerables.

Node.js cumple con este requisito, capaz de recibir todas las peticiones y realizar el envío de datos en el plazo esperado.

**Renderizado:** El servidor debe de renderizar todas las figuras de los clientes concurrentemente, siendo una operación considerablemente costosa y se ejecuta cada vez que el cliente realiza un cambio en su figura o el ángulo de visión.

Debido a que Node.js no es capaz de afrontar estas operaciones, la opción tomada es delegar este proceso al BackEnd. Tomando esta medida, Node.js también puede realizar la función de balanceador de carga entre las máquinas dedicadas al cálculo.

**Servir la página al cliente:** Como medida para incrementar la robustez de la aplicación, es adecuado separar el FrontEnd del editor 3D del servidor web destinado a enviar el código de la página al cliente. Este servidor de código podría también implementarse en Node.js o en cualquier otra tecnología sin repercutir en el resto de la aplicación o en la experiencia del usuario.

Como conclusión final se considera que Node.js es completamente adecuado para ciertas partes de la aplicación, como el FrontEnd, que con otra tecnología sería demasiado complejo construirlas obteniendo las mismas prestaciones. Para el resto del sistema, Node.js es completamente desaconsejable (para el renderizado).

### 8.2.2. durchblicker.at

Es un comparador de precios alojado en <https://durchblicker.at>. Básicamente consiste en una base de datos de ofertas, el cliente selecciona unas características se devuelven los resultados al cliente.

Los requisitos de esta aplicación son pocos:

**Permitir consultas simultáneas a la base de datos:** Hay un gran volumen de clientes realizando consultas y actualizando datos de proveedores, por lo que el acceso a la base de datos se convierte en un punto crítico. Por lo general cualquier tecnología de desarrollo web ofrece un buen acceso a la base de datos, por lo que el uso de Node.js en este aspecto es indiferente.

**Servidor web:** Para que los clientes puedan realizar las consultas, es necesario que previamente carguen la página de la aplicación. Node.js es un buen servidor web bajo el inconveniente de la dificultad del desarrollo de páginas generadas en el servidor por JavaScript. No es indispensable para la aplicación ser construido en Node.js.

Con este análisis observamos que la aplicación puede ser construida íntegramente con el entorno, aunque también podría ser construido con cualquier otra tecnología.

### 8.2.3. loggly

El negocio de **Loggly** <https://www.loggly.com/>consiste en la recopilación de registros de los de clientes mediante un sistema basado en la nube. Las máquinas de los clientes envían los registros cifrados a los servidores de **loggly** y estos los capturan, almacenan y tratan la información, los registros recopilados pueden ser de los Sistemas Operativos, de programas, etc. Los requisitos el sistema serian:

**Seguridad:** Los datos registrados de los clientes son considerablemente sensibles, por lo que es necesario verificar desde todos los ámbitos seguridad de la aplicación.

Las garantías de seguridad que ofrece Node.js son suficientes para la aplicación, el posible riesgo surge con los módulos publicados por terceros, los cuales deben ser analizados antes de su uso, mientras que con otros frameworks con funciones y contenido enteramente propios este problema no existiría. Si se escoge Node.js para el desarrollo se deben de evitar los módulos externos.

**Recepción de registros:** El tamaño de los registros que se transmiten son generalmente pequeños, aun así, unos pocos clientes pueden generar un tráfico considerable. Node.js es un entorno ideal para este requerimiento puesto que ha sido diseñado para este propósito y es totalmente recomendado.

**Descifrar mensajes:** Los registros deben de ser cifrados para garantizar la confidencialidad del mensaje. Una vez recogidos en el servidor, deben de descifrarse para poder ser procesados y almacenados.

Node.js puede descifrar los mensajes con bastante velocidad bajo el inconveniente de que durante la operación se desatienda la captura de mensajes, por ello es

recomendable el modo clúster. Esta operación puede ser realizada por diversos lenguajes con resultados de benchmarking similares a los de Node.js.

Se puede concluir que la aplicación puede ser construida sin problemas con cualquier framework, aun así con Node.js se obtienen mejoras en la recopilación de datos de los clientes.

### 8.2.4. dnode

**dnode** es un sistema RCP que permite a los programas la ejecución de código en máquinas remotas siendo transparente la fase de comunicación, por lo que el programa debe de comunicar programas locales en la maquina con maquias remotas.

Para el correcto funcionamiento se necesitan los siguientes requisitos:

**Comunicación con múltiples máquinas:** La ampliación no necesita más que servir de interfaz entre máquinas y abstraer la red por lo que es necesario una ágil comunicación. Debido a que el principal propósito de Node.js es la comunicación entre máquinas, esta es una tecnología ideal.

**Comunicación con programas locales:** Para servir de interfaz con otros programas es imprescindible una ágil comunicación con ellos. Node.js gestiona las comunicaciones locales con la misma metodología que con las comunicaciones por red, por lo que es igualmente eficiente para este propósito.

Los requisitos de esta aplicación son pocos, pero se basan principalmente en la comunicación y en la gestión de entradas y salídas. Debido a que Node.js ha sido expresamente diseñado para cumplir con estos objetivos, se convierte en una aplicación idónea.

## 8.3. Aplicación “Node-censo”

Se ha desarrollado una aplicación (**Node-censo**) con la que poder comparar sus características con las posibilidades que ofrece Node.js. La aplicación consiste en un acceso web a una base de datos, que cuando recibe instrucciones de los clientes, se ejecuta una pequeña lógica (principalmente de validación de datos) y se registran en la DDBB.

Los requisitos de la aplicación son:

**Comunicaciones cortas:** La prioridad de la aplicación es mantener las comunicaciones lo más cortas posibles, los clientes envían las instrucciones al servidor, y en cuanto han sido capturadas, la comunicación finaliza sin la necesidad de que la operación sobre la DDBB se complete. Con esto se pretende que la aplicación atienda a una gran cantidad de usuarios.

**Minimizar la lógica:** Observamos que la pequeña cantidad de lógica no es un obstáculo y el tiempo de proceso de cada petición no es significativo, ya que la petición termina antes de completar la lógica, que con otros lenguajes clásicos sería muy difícil de implementar.

**Separar los canales:** Otra ventaja única de Node.js es poder elegir el puerto en que se evitan paquetes. No es un requisito indispensable y podrían haberse realizado todas las comunicaciones por el mismo puerto 80, pero el motivo de tomar esta decisión es separar los tipos de comunicación por puertos y como ejemplo ilustrativo de las capacidades de Node.js.

**Mantener sesiones:** La gestión de sesiones de usuarios es un poco más complejo, ya que es necesario la instalación de un módulo como puede ser "**sesion**", el cual añade la funcionalidad del registrar sesiones en Node.js. Puede ser usado cualquier otro modulo por lo que se puede derivar a múltiples formas de inicio de sesión según el modulo escogido habiéndose ya nombrado los problemas e inconvenientes que genera.

En otros lenguajes como PHP, sin tener en cuenta la lógica de validación, la generación de la sesión solo consta de una instrucción siendo un paso muy transparente para el programador, mientras que con Node.js se debe de encontrar un módulo que mejor se adapte e investigar sus funciones.

Con el apartado anterior podemos obtener la conclusión de que Node.js es una buena tecnología para el desarrollado de aplicaciones de servidor. Es especialmente útil para comunicarse con otras máquinas y dispositivos, empleando un servidor menos potente que con otra tecnología.

En aplicaciones como la desarrollada para este proyecto, se necesita comunicar datos constantemente entre cliente-servidor bidireccionalmente y que la ejecución sea asíncrona, características que la mayoría de lenguajes no son capaces de ofrecer, como ejemplo PHP, que necesitaría recargar la página entera para actualizar los datos del cliente y la ejecución sería síncrona manteniendo un hilo bloqueado innecesariamente por el cliente hasta completar la operación en la base de datos, en lugar de liberar al cliente inmediatamente después de recibir la petición.

La experiencia final del usuario se traduce en poco tiempo entre operaciones y un constante refresco de datos.

La construcción y mantenimiento de la interfaz web es bastante complejo con el uso de JavaScript, no ha sido diseñados para la generación de páginas, como otras tecnologías en las que si como PHP o Ruby on Rails. En **Node-censo** este inconveniente ha sido resuelto enviando al cliente una plantilla estática HTML y el resto de la página se genera en el cliente mediante JavaScript, reduciendo la carga sobre el servidor. Aun así, sigue siendo un sistema complejo.

## 8.4. Clasificación de aplicaciones

Tomando el anterior conjunto de aplicaciones, se pueden dividir en tres categorías según su idoneidad para ser implementadas en Node.js. Las tres categorías son:

**Aplicaciones adecuadas**: Consisten en aquellas en las que el entorno de Node.js es significativamente beneficioso para las características del programa. Por ejemplo, en un servidor proxy en que el tráfico es constante, la característica de comunicación con multitud de máquinas concurrentes es imprescindible.

**Aplicaciones intermedias:** Son aquellas en las que el entorno no ofrece ninguna ventaja frene a otros frameworks, como podría ser un pinger, un programa que lanza ping a dispositivos de red para comprobar su conexión. Los requisitos de este programa son escasos y podría desarrollarse con Node.js, como un script o con C.

**Aplicaciones contraproducentes:** A diferencia de las aplicaciones adecuadas, este tipo de aplicaciones Node.js repercute negativamente en las características de este. Como ejemplo de este tipo de aplicaciones están los simuladores en los que el cliente envía datos y el servidor los procesa. En este tipo de aplicación sería contraproducente Node.js, pues la simulación bloquearía el único hilo en ejecución.

# 9. Validación de la Propuesta

Durante el Marco conceptual y otros puntos del proyecto, se ha visto que Node.js tiene aspectos muy útiles y otros contraproducentes según la tarea que se realice. A continuación se muestran tres aplicaciones, una que aprovecha adecuadamente las ventajas del entorno, otra que su empleo es contraproducente y por ultimo otra aplicación que no se ve afectada ni positiva ni negativamente.

## 9.1. Aplicación del Marco

Las aplicaciones escogidas para ser analizadas son tres, una por cada tipo de aplicación, para la aplicación recomendable se ha escogido un programa de videoconferencia entre dos clientes **PeerChat** habiendo muchos ejemplos de aplicaciones recomendables de este tipo.

Para la aplicación contraproducente se ha escogido **word-finder**. Es muy difícil encontrar aplicaciones con estas características debido a que todas se diseñan pasa ser positivas.

Por último encontramos dos aplicaciones intermedias como **sortis** y **e-resistible**, que no se valen de las prestaciones de Node.js ni tiene una repercusión negativa. De esta última solo realizamos mención de su existencia.

### 9.1.1. Aplicación adecuada

Aplicación: **PeerChat.**

Descripción: Consiste en un chat donde el los clientes pueden abrir salas privadas de dos participantes con videoconferencia. Si se incluye un tercer participante se podrá unir al chat de texto pero no a la videoconferencia, el código es libre para poder adaptarse a otras necesidades.

Dirección Web: <http://peerchat.net/>.

Dirección del código: <https://github.com/Hironate/PeerChat>.

Tipo de aplicación: Chat con videoconferencia.

Arquitectura: Se basa en un modelo muy simple de un servidor y varios clientes en grupos de dos. La función del servidor es servir la página con la lógica del cliente y en difundir datos entre los respectivos clientes, el servidor espera a que un cliente solicite crear una sala y un segundo cliente tendría la posibilidad de unirse a la sala creada o crear una nueva. Las salas creadas se comunican con los clientes mediante sockets.

Cuando un cliente envía datos de texto, audio o video, estos se envían al servidor y se difunden entre todos los participantes de la sala (por defecto son dos).

Características: Es una aplicación ideal para Node.js ya que cumple con las propiedades para las que Node.js ha sido diseñado como la gestión de comunicaciones por red. Las desventajas del framework han sido bien enmascaradas en la aplicación, como la dificultad de la generación de páginas HTML, solucionándolo con una pequeña página estática para cualquier uso. Node.js tampoco puede efectuar cálculo de datos rápidamente, los datos se reenvían inmediatamente después de ser recibidos, por lo que la lógica del servidor es despreciable.

Si esta aplicación hubiera sido construida con otra tecnología, cada cliente o sala activa consumiría un hilo entero del servidor, reduciendo los recursos para el resto de comunicaciones. La creación de los sockets para esta aplicación es más sencilla que usando Java.

### 9.1.2. Aplicación intermedia

Son dos las aplicaciones neutras encontradas como ejemplo. La primera es **appetise** <www.e-resistible.co.uk>, consiste en un comparador web de restaurantes, el código fuente no es público, aun así se intuye que la comunicación es petición-respuesta, el cliente inserta un código postal y el servidor mediante una consulta a una base de datos devuelve una lista de comercios cercanos.

La otra aplicación es **sortis**, es la elegida para el análisis debido a que su código es público.

Aplicación: **sortis.**

Descripción: Trata de una aplicación empleada para la gestión de tweets, la aplicación los recoge y ordena por favoritos. El autor lo ha construido como plantilla para crear una aplicación más potente.

Dirección del código: <https://github.com/amirrajan/sortis>.

Tipo de aplicación: Gestor de tweets.

Arquitectura: Consta de tres partes, el servidor, el cliente y el servidor de tweets. El cliente se conecta con el servidor **sortis**, a través del cual se conecta con el servidor de twees mediante de un largo proceso de autentificación por parte del usuario, el servidor **sortis** recoge los tweets, los gestiona conforme este programado y los devuelve al cliente.

Características: la aplicación podría haber sido construida con cualquier otra tecnología sin que esta se vea afectada, el volumen de comunicaciones es muy pequeño, lo que no aprovecha la capacidad de Node.js de gestión de comunicación. La generación de página HTML con los tweets es muy simple, y la lógica del servidor es pequeña, pero depende de la cantidad de datos que deba gestionar, por ejemplo la ordenación de un conjunto de tweets, por lo general estos no generan un volumen de datos excesivos.

### 9.1.3. Aplicación contraproducente

Aplicación: **Word-finder.**

Descripción: Aplicación que devuelve al cliente una lista de palabas a partir de fragmentos dados. Es útil para encontrar palabras que rimen entre sí.

Dirección del código: <https://github.com/amirrajan/word-finder>.

Tipo de aplicación: Buscador de palabras a partir de fragmentos.

Arquitectura: Los clientes envían al servidor fragmentos de palabras, este los recoge y los compara con todas las palabras contenidas en un array. Las que hacen matching se almacenan en un array auxiliar que es devuelto al cliente.

Características: Esta es un ejemplo de una aplicación totalmente no recomendada para ser implementada en Node.js, la comunicación con el cliente es petición-respuesta y el volumen de peticiones es bastante escaso desaprovechando las propiedades de entrada y salida de Node.js. La búsqueda de las palabras las realiza el propio Node.js en un array de más de 200.000 elementos, provocado la monopolización del único hilo en recorrer ese array. El diseño ideal sería almacenar las palabras en una base de datos y las búsquedas se realizarían mediante una consulta, liberando al hilo de realizar a cotosa operación.

## 9.2. Desarrollo de una Aplicación "Node-censo"

Se va a desarrollar una aplicación que cumpla las propiedades de Node.js. Para ello se ha pensado un algo tan útil como la interfaz web de una base de datos de un censo. La aplicación será identificada como "**Node-censo**".

Se implementará el nuevo paradigma de comunicaciones, donde la interfaz mantenga y los datos se actualicen dinámicamente.

Esta base de datos debe de permitir las operaciones básicas como añadir, borrar, buscar y modificar.

Las tecnologías usadas para la creación del proyecto han sido JavaScript, HTML, MySQL y Node.js. El servidor está estructurado en tres módulos y la base de datos y cada uno de ellos realiza la siguiente función:

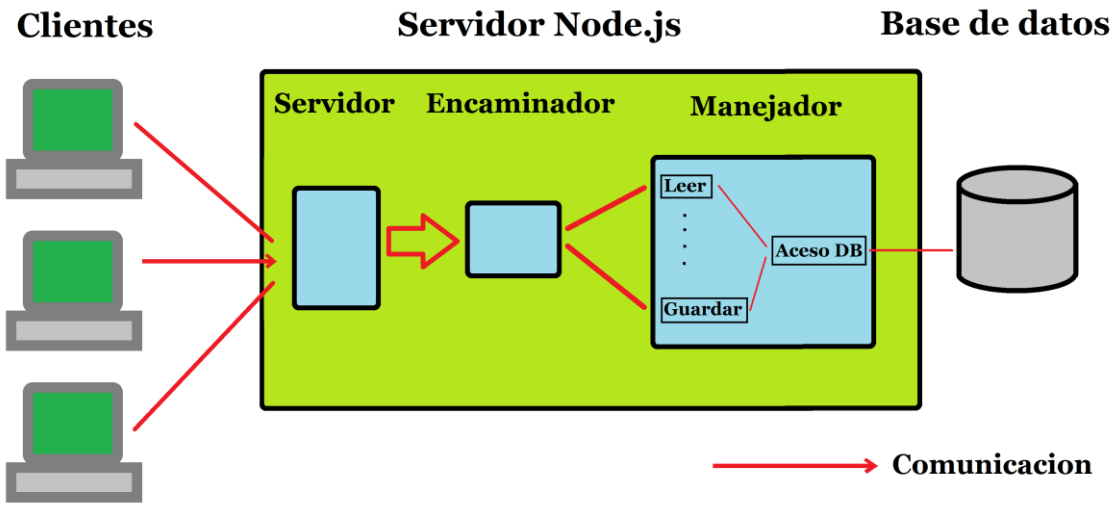


*Figura 16 - Estructura de Node-censo*

**Servidor:** Es el módulo encargado de la comunicación, recibe los mensaje de los clientes devuelve respuestas. Las instrucciones recibidas son enviadas al Encaminador. El puerto en el que escucha es el 5000 y mantiene comunicación por HTTP, en cuanto recibe los datos, llama al encaminador con la ruta pedida por el cliente como parámetro.

**Encaminador:** Este módulo se encarga de recibir la petición del servidor y dirigirla a la función correspondiente del manejador, (si no existe una función para la petición se devuelve un error). Recibe la ruta pedida por el cliente como un parámetro, lo evalúa y llama a la función correspondiente de esa ruta en el manejador.

**Manejador:** El módulo consiste en una colección de funciones que tratan las distintas peticiones que puede hacer un cliente y el acceso la base de datos. Estas funciones se dividen en 3 grupos:

Las funciones para servir la página.

Las funciones para interpretar las peticiones de los clientes en formato SQL.

Las funciones para evitar repetición de código como acceder a la base de datos o enviar información al cliente.

La página servida consiste en un fichero HTML, un fichero en JavaScript y un CSS, además de algunas imágenes como el favicon.ico o el fondo. La decisión de obtener el HTML mediante fichero era provisional, ya que la idea original era que el servidor lo generara en una función aparte con los métodos propio de JavaScript a modo de demostración, pero se dejó así por la simplicidad de tener un fichero con todo el código visible para depurar mejor. La opción elegida, es enviar el código para generar la página en el cliente, restando carga al servidor.

Por parte del cliente, solo se pide la página una vez al servidor, junto con el resto del código JavaScript, a partir de aquí, la página cargada siempre es la misma, y todas las demás comunicaciones se realizan mediante JavaScript, previniendo el refresco de página, pues el contenido de las variables de los clientes se perdería.

En resumen, es una aplicación idónea, no se ejecutan grandes instrucciones sino que solo se gestiona la entrada y salida. Cuando se recibe una instrucción, la función correspondiente convierte los datos recibidos a una instrucción SQL y la envía a la base de datos.

La parte del cliente está desarrollada siguiendo dos versiones, una sin JavaScript que reacciona con el modelo de Petición-Respuesta, y otra con JavaScript más dinámica y con refresco de datos automático. La versión sin JavaScript está destinada al testeo del servidor, mientras que la contingente de JavaScript está destinada como interfaz final.

El refresco de los datos se puede realizar de tres formas, o un bucle activo en el cliente que refresque la consulta realizada, que el servidor notifique al cliente que se ha realizado un cambio y el cliente relance la consulta o que el servidor conozca los clientes activos y sus consultas, y en cuanto se realice un cambio, reenvié el nuevo resultado de la consulta realizada al cliente.

El método elegido dependería del volumen de clientes y del tamaño de la base de datos. Con una base de datos pequeña, el cliente podría tener la tabla entera y mostrar los datos que necesite, si existen muchos clientes, la solución más adecuada sería enviar la modificación a los clientes.

# 10. Comparación y Evidencias

Este apartado está destinado a comparar el entorno de Node.js con otros similares con el fin de comprender las ventajas que ofrece esta tecnología y como podría responder la máquina que lo soporta. Las aplicaciones construidas también se compararán entre sí para analizar el comportamiento según el diseño de la aplicación.

## 10.1. Comparación de tecnologías

A continuación se exponen las pruebas realizadas en <http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php>, con los siguientes resultados:

Se realizaron pruebas comparando un servidor Apache con PHP y otro realizado con Node.js, con el fin de analizar los recursos del sistema consumidos por ambas tecnologías. La prueba consistía en servir una página web con el contenido "Hello World" a una distinta cantidad de clientes. En la primera prueba se envían 100.000 peticiones con un nivel de concurrencia de 1000 peticiones y en la segunda 1.000.000 peticiones con un nivel de concurrencia de 20.000 peticiones.

Los resultado extraídos en las pruebas son el tiempo empleado en responder a todas las peticiones, la cantidad de peticiones fallidas, el consumo de memoria y el de CPU.

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

http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<p>Hello World</p>');
  res.end();
}).listen(8080);
```

*Figura 17 - Código de prueba en Node.js*

```
<?php
echo '<p>Hello World</p>';
?>
```

*Figura 18 - Código de prueba en PHP*

**Resultados de la primera prueba:**

| | Node.js | Apache + PHP |
|---|---|---|
| Tiempo empleado (segundos) | 21.162 | 121.451 |
| Peticiones fallidas | 147 | 879 |

*Figura 19 - Resultados de la prueba 1*

En la tabla de resultados de la primera prueba observamos que el tiempo necesario para atender a las peticiones usando PHP es 6 veces mayor que con Node.js, igualmente el número de peticiones fallidas se incrementa 6 veces más.

En cuanto al consumo de CPU, el porcentaje de uso es prácticamente idéntico, el cambio significativo se encuentra en el tiempo de uso. Si el porcentaje usado es el mismo, pero el tiempo necesario es menor, implica que el consumo por petición de cliente en Node.js es menor que en Apache con PHP.

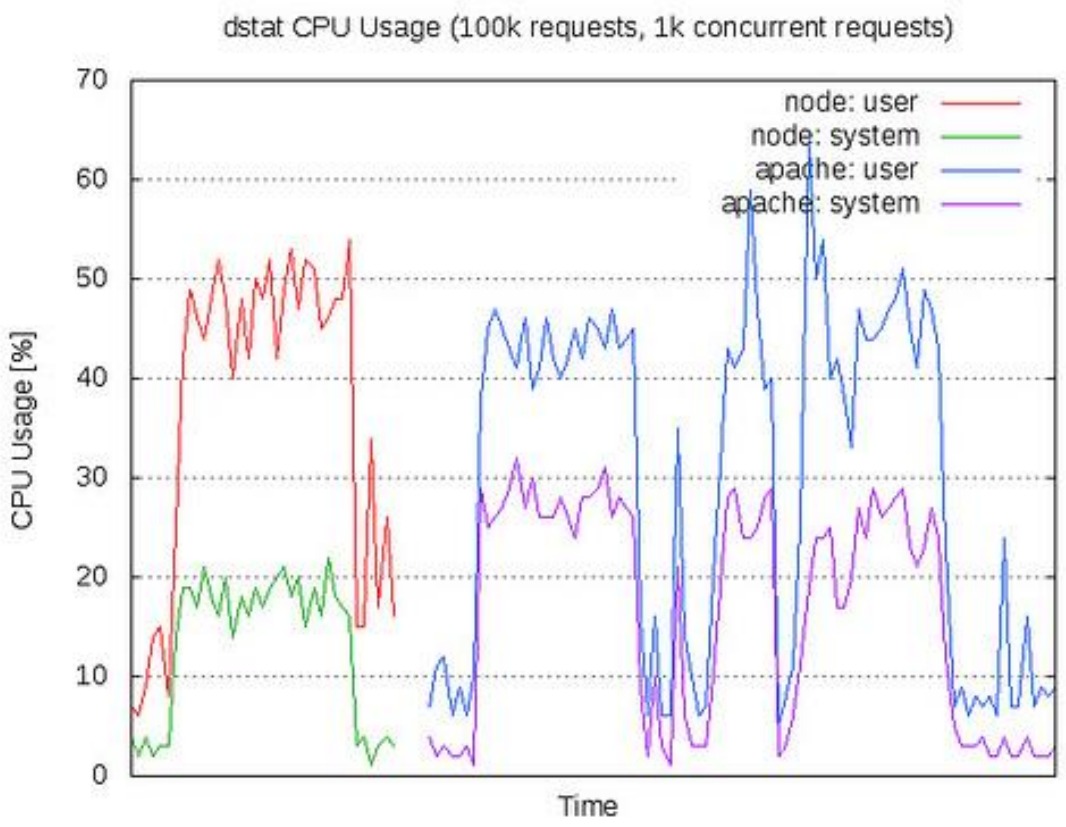


*Figura 20 - CPU consumida en la prueba 1*

En el grafico de memoria, damos cuenta del aumento del uso de la memoria en Node.js. En relacion a la memoria consumida por Apache, se puede considerar que Node.js continua necesitando menos recursos debido a que el aumento de meroria requerida es mucho menor que la memoria consumida por Apache durante el tiempo extra desde que Node.js termina hasta que Apache termina.

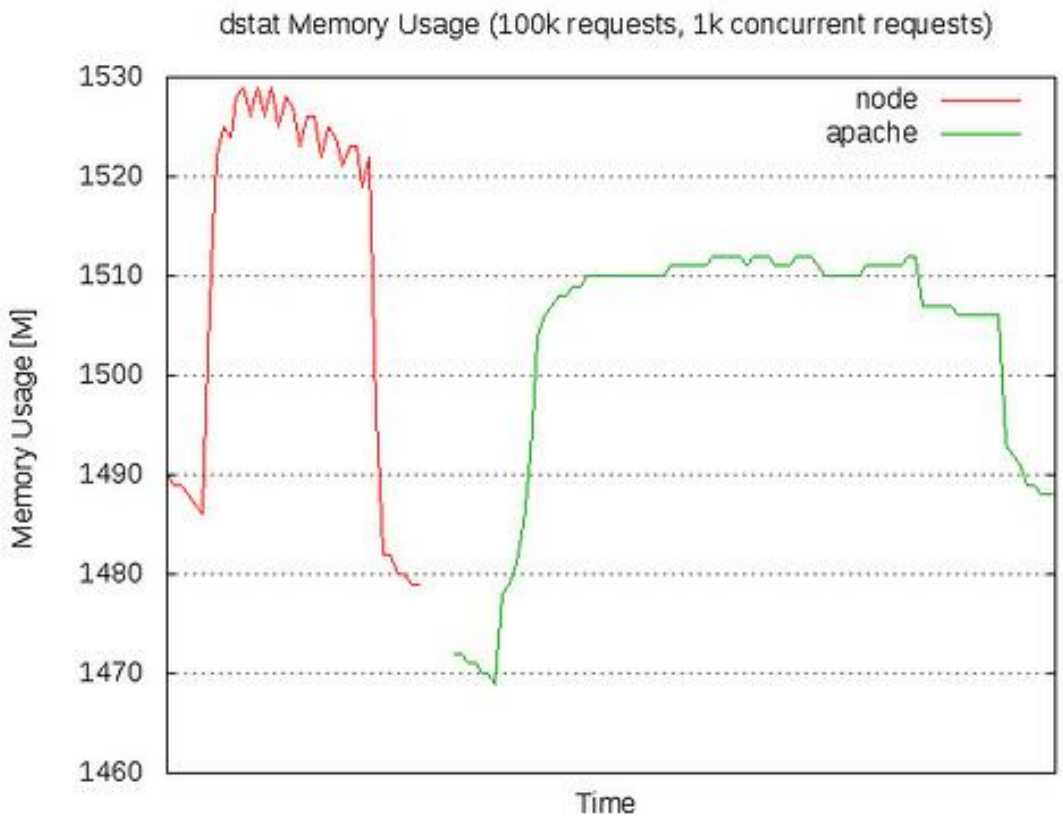


*Figura 21 - Memoria consumida en la prueba 1*

**Resultados de la segunda prueba:**

| | Node.js | Apache + PHP |
|---|---|---|
| Tiempo empleado (segundos) | 1043.076 | 3570.753 |
| Peticiones fallidas | 25227 | 2617614 |

*Tabla 22 - Resultados de la prueba 2*

En esta segunda prueba, el tiempo empleado con Node.js sigue siendo menor que con Apache, siendo la diferencia menos significativa en 3,5 veces más rápido. Node.js sigue siendo más fiable que la otra tecnología cometiendo 106 veces menos peticiones fallidas.

En cuanto a los recursos consumidos, el uso de memoria y CPU, se obtienen resultados idénticos a la anterior prueba. Por parte de Node.js, el uso de CPU ha aumentado un 20% más respecto a Apache. Este aumento es justificado con la reducción de 3 veces el tiempo, por lo que el uso de Node.js es una opción más productiva. La misma conclusión se puede obtener con los datos recogidos con la memoria utilizada.

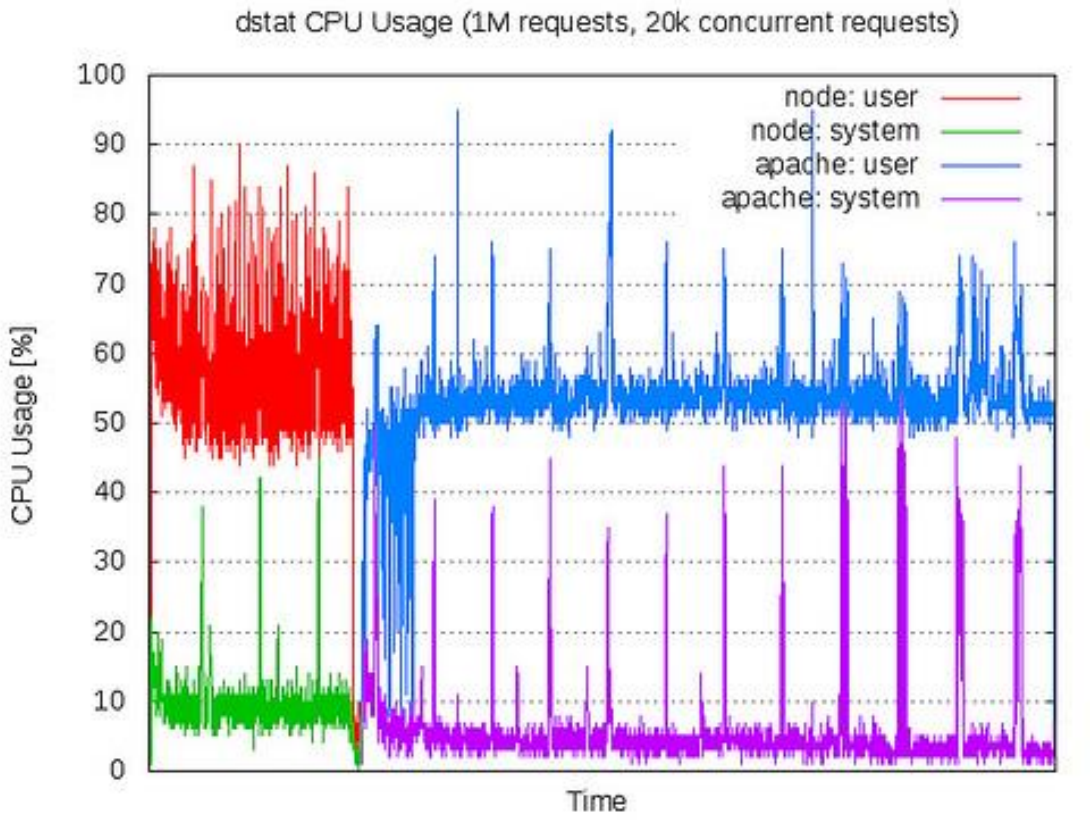


*Figura 23 - CPU consumida en la prueba 2*

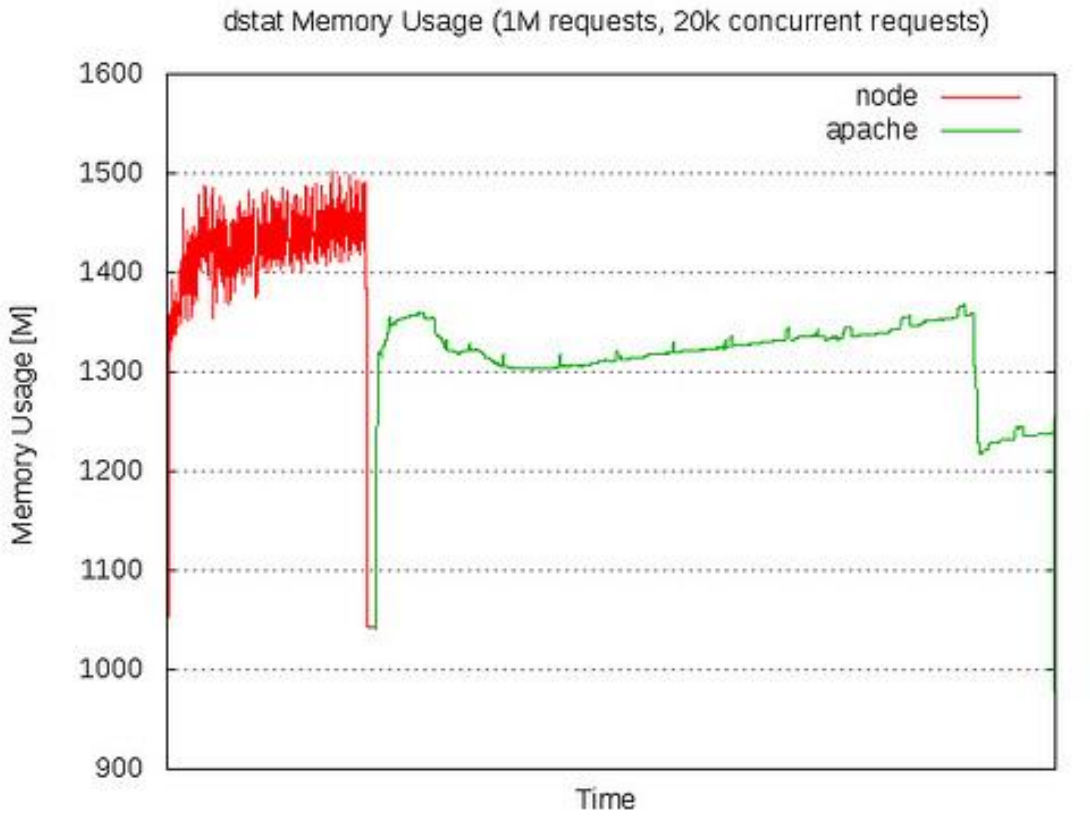


*Figura 24 - Memoria consumida en la prueba 2*

Gracias a estas pruebas, obtenemos que Node.js ofrece unas grandes prestaciones en la resolución de peticiones de clientes y en el aprovechamiento de los recursos de la máquina.

A diferencia de otros programas, la potencia de Node.js no se encuentra limitada por la cantidad de hilos asignados (en un caso general solo necesita uno), al delegar cualquier tipo de cálculo u obtención de recursos al sistema, el límite de potencia se encuentra en la capacidad del sistema en obtenerlos. Como consecuencia de este modelo, Node.js puede generar picos de demanda de recursos que pueden afectar a otras aplicaciones alojadas en la máquina. Para prevenir estos picos de carga se puede limitar la cantidad de tareas admitidas por POSIX,  o emplear otras tecnologías más constantes en el consumo de recursos.

Realizando un balance general entre las dos aplicaciones, Node.js demuestra ser una excelente tecnología, demostrando que en aplicaciones con gran demanda de comunicación con poca lógica, la recepción de peticiones puede dejar de ser el cuello de botella del servidor, cumpliendo su objetivo de gestión de entradas y salidas.

## 10.2. Comparación entre aplicaciones

A continuación se tomará un conjunto de aplicaciones con características similares y se comparará su desarrollo.

Se puede observar que **BrowserQuest**, **Clara.io** y **PeerChat** son de las aplicaciones que mejor aprovecha las propiedades de Node.js, atendiendo una gran cantidad de comunicaciones con los clientes mediante mensajes abundantes y de poco contenido.

**BrowserQuest** y **PeerChat** están íntegramente construidas con Node.js, su funcionamiento está basado en la transmisión de mensajes y no requieren de lógica, mientras que **Clara.io** tiene un funcionamiento basado en el cómputo, por lo que diseñar la aplicación íntegramente en Node.js sería considerado un error y se estaría realizando una tarea completamente desaconsejada que el entorno de programación no sería capaz de asumir, como el renderizado 3D. Este sería el caso de **Word-finder**, aplicación totalmente construida con Node.js en lugar de estructurarse y adoptar la tecnología adecuada, por lo que la aplicación tiene un gran problema de escalabilidad funcionando bien con pocos clientes hasta que estos aumentan y bloquean el hilo de Node.js.

**Chess Hub** necesita una comunicación síncrona con los clientes, solo se permite la comunicación entrante o saliente en determinados momentos para cumplir con las normas del juego, en los demás aspectos, es muy similar a **BrowserQuest** debido a que también es una aplicación centrada en las comunicaciones y totalmente construido en Node.js. Anteriormente tenía un modo de un único jugador con lógica implementada también en Node.js, este modo se encuentra deshabilitado en el servidor debido a que la lógica de un solo jugador o puede ser asumido por el entorno, siendo una posible solución implementar la lógica en otro framework.

**Durchblicker**, **HabirRPG** y **loggy** son aplicaciones enfocadas en el registro y tratamiento de datos, consistiendo básicamente en una interfaz de comunicación que actúa sobre una base de datos, dependiendo de la aplicación, estas pueden contener lógica que actúe sobre esos datos (validación, comparación, etc). La ganancia de usar Node.js en ese tipo de aplicaciones se encuentra en atender a gran cantidad de clientes y en actuar sobre bases de datos de respuesta lenta (debido a la cantidad de datos, acceso remoto, potencia de hardware, etc).

En contrapartida, **Open Marriage** es un sencillo servidor capaz de ofrecer una página estática, diseñado para atender a un centenar de clientes en total. Aunque está muy bien hecha, no aprovecha las propiedades de Node.js y un servidor Apache podría desempeñar mejor esa función.

## 10.3. Comparación de Node-censo

En este apartado se comparará el programa desarrollado para este proyecto **"Node-censo"** con algunos de los programas analizados.

Registrar operaciones como lo hace el programa es una operación idónea para Node.js, tiene mucho en común con **Durchblicker**, **HabirRPG** y **loggy**, pues también consisten en ofrecer una interfaz de la base de datos con los clientes. Una importante mejora para **Node-censo** sería establecer sesiones con los clientes y validad sus operaciones según permisos. Otra de las similitudes con **HabirRPG** es el inmediato refresco de los datos en cuanto se realiza un cambio en el progreso del cliente. Como diferencia de las anteriores aplicaciones, **Node-censo** no trata los elementos de la base de datos, tan solo captura y muestra los datos a los clientes sin ningún proceso de comparación ni ordenación.

La aplicación presenta un cierto grado de sencillez para realizar una función ilustrativa y ayudar a comprender mejor la estructura, se ha querido estructurar bien la aplicación para servir de ejemplo a futuros desarrolladores.

Comparando con otras aplicaciones como **BrowserQuest**, el censo desarrollado no refleja las operaciones de un cliente hacia el resto, lo que se transmite son los cambios en la base de datos.

En cuanto a la seguridad, se ha incluido el menor número de módulos posible para maximizar la seguridad y la cantidad de código propio. Y se validan todos los datos de entrada para prevenir la inyección de código SQL sobre la base de datos o sobre el propio código del servidor en JavaScript. Estas validaciones solo están implementadas en el servidor, mientras que en el lado del cliente solo se limitan los campos del formulario debido a que un usuario experto podría anular fácilmente estas verificaciones tanto por JavaScript como por el tipo de entrada de formulario.

Como todos los requisitos de la aplicación son adecuados para Node.js, la integridad de la aplicación ha sido construida con esta tecnología.

# 11. Conclusiones y trabajo futuro

## 11.1. Valoración Personal

Gracias a este trabajo he aprendido a comparar tecnologías y a como planificar un proyecto antes de lanzarme a programar. Me he dado cuenta de primera mano que el análisis previo antes de realizar un trabajo puede influir enormemente en el resultado final.

Me he sorprendido de las posibilidades que ofrece Node.js, de la que conocía su existencia pero desconocía sus capacidades, también he aprendido a no sobrevalorar recursos y a aprender a analizar los inconvenientes, tarea bastante difícil cuando tu percepción personal del recurso es buena.

Ha sido especialmente gratificante desarrollar una aplicación, aprender sobre esta nueva tecnología y poder documentar mis conclusiones y conocimientos adquiridos para que puedan ser aprovechados por otros desarrolladores.

## 11.2. Valoración Objetiva

Aunque es una tecnología joven, Node.js comienza ganar popularidad y ya existen ejemplos del aumento de rendimiento al migrar a esta nueva tecnología.

Es un buen gestor de entradas y salidas, su misión principal es delegar trabajo y el límite de peticiones a atender no está limitado por el número de hilos (comúnmente en los servidores convencionales esto es un problema) convirtiéndolo en una eficaz solución FrontEnd.

Ofrece una gran flexibilidad de comunicación entre clientes y servidores al utilizar el mismo lenguaje (JavaScript), no es necesaria ninguna conversión de datos o instrucciones.

Abre un nuevo paradigma entre la comunicación de clientes y servidores, (no es obligatorio el modelo de petición-respuesta) permitiendo una mayor interacción entre máquinas.

En contrapartida, se necesita esfuerzo para crear un servidor desde cero y comprender su forma de trabajar para aprovechar sus propiedades, especialmente cuando la formación sobre la programación asíncrona es escasa.

Una programación ineficaz puede bloquear el único hilo, por lo que hay que centrarse en evitar este problema, además de que con un solo hilo no se aprovechan

todos los recursos de la máquina. Un programador iniciado en la programación asíncrona aun conociendo JavaScript le resultaría bastante difícil construir una aplicación.

La mayoría de módulos publicados ofrecidos por terceros son bastante inmaduros, pero ayudan a ampliar la comunidad de Node.js.

## 11.3. Trabajo Futuro

Como futuros trabajos, se propone el desarrollo de compatibilidad con PHP para abrir un nuevo mercado en hosting a Node.js, ya que migrar páginas entre servidores clásicos con Node.js presenta un grave problema y ofrecería una mayor flexibilidad al programador.

También se propone la investigación de la programación asíncrona, indicando sus propiedades y desventaja e instruyendo para comprender su funcionamiento y prevenir evidentes errores.

Otro gran trabajo futuro sería la realización y publicación de modelos de servidores básicos como HTTP o SMTP y así disponer de una plantilla sobre la que construir los servidores más usuales, reduciendo el tiempo de desarrollo de un servidor y disponiendo de una base ya probada y segura.

# 12. Referencias bibliográficas

TOMISLAV CAPAN Why The Hell Would I Use Node.js? A Case-by-Case Tutorial <*http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js*> [Consulta: Febrero de 2015]

HASENJ Why I think Node.JS is a terrible plataform <*https://news.ycombinator.com/item?id=4495101*> [Consulta: Febrero de 2015]

ROUYU SUN Node.js is Perfectly Fine And Probably You Don't Need it <*http://ruoyusun.com/2013/03/31/node-js-is-perfectly-fine-and-probably-you-dont-need-it.html*> [Consulta: Febrero de 2015]

Stack Overflow How to decide when to use Node.js? <*http://stackoverflow.com/questions/5062614/how-to-decide-when-to-use-node-js*> [Consulta: Febrero de 2015]

FELIX GEISENDÖRFER Felix's Node.js Convincing the boss guide <*http://nodeguide.com/convincing_the_boss.html*> [Consulta: Febrero de 2015]

FELIX GEISENDÖRFER Understanding node.js <*http://debuggable.com/posts/understanding-node-js:4bd98440-45e4-4a9a-8ef7-0f7ecbdd56cb*> [Consulta: Febrero de 2015]

STOIMEN Diving into Node.js – Introduction & Installation <*http://www.stoimen.com/blog/2010/11/16/diving-into-node-js-introduction-and-installation/*> [Consulta: Febrero de 2015]

ÓSCAR RAY Una docena de conceptos que deberías conocer sobre Node.js <*http://unadocenade.com/una-docena-de-conceptos-que-deberias-conocer-node-js/*> [Consulta: Febrero de 2015]

BETABEERS ¿Qué usos reales le dais a node.js? <*https://betabeers.com/forum/que-usos-reales-le-dais-a-nodejs-25/*> [Consulta: Febrero de 2015]

AEX R. YOUNG Let's Make a Web App: Nodepad <*http://dailyjs.com/2010/11/01/node-tutorial/*> [Consulta: Febrero de 2015]

MANUEL KIESSLING El Libro para Principiantes en Node.js <*http://www.nodebeginner.org/index-es.html*> [Consulta: Marzo de 2015]

GIT HUB Fuente de código de proyectos Open Source. <https://github.com/> [Consulta: Febrero, Marzo y Abril de 2015]

GODTIC Se habla del uso de Node.js, activación y aplicaciones del modo clúster. <*http://www.godtic.com/blog/2013/11/12/node-js-rapido-como-el-rayo/*> <*http://www.godtic.com/blog/2013/07/27/modo-cluster-para-node-js/*> [Consulta: Marzo de 2015] [Actualmente cerrada]