Explotación de servicios gratuitos Cloud Computing

Jordi Mahiques Alamo

Índice

Introducción

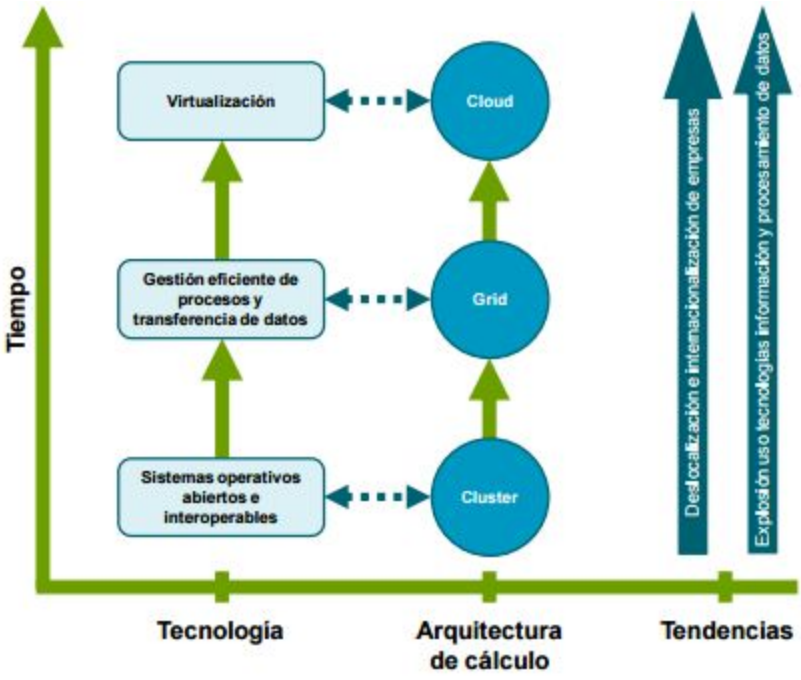
Antecedentes

En las últimas décadas los procesos de deslocalización e internacionalización de las grandes empresas, unidos a la explosión en el uso la de tecnologías de información y procesamiento de datos, han hecho que las necesidades de cómputo de las grandes empresas y organizaciones hayan crecido a un ritmo superior al que lo hacía la capacidad de cálculo de los ordenadores personales. Por este motivo, y para satisfacer las necesidades de los sistemas de computación más exigentes, se ha producido una interesante evolución de las arquitecturas de cálculo, basada fundamentalmente en la ejecución simultánea de procesos en múltiples equipos informáticos.

Debido a las necesidades de cómputo descritas, se ha venido realizando un importante esfuerzo en la investigación de capacidades para la ejecución de procesos en múltiples computadores. Esta tendencia fue impulsada originalmente por la utilización de sistemas abiertos, interoperables y protocolos de comunicación estándar que permitían la comunicación eficiente entre sistemas y tecnologías heterogéneos. El primer paso de esta evolución fue en gran medida propiciado por los sistemas operativos tipo Unix que permitieron la configuración de clusters, es decir, agrupaciones de ordenadores con componentes de hardware comunes que se comportan como un único computador.

Tras varias décadas de investigaciones y desarrollos en estas tecnologías, la irrupción del sistema operativo Linux y sus estándares abiertos permitió implementar clusters basados en la arquitectura estándar de los PC, consiguiendo instalaciones de cálculo de alto rendimiento a bajos precios y popularizando esta solución durante la década de 1990 . Estos clusters sufrieron un proceso de especialización para proporcionar servicios de cálculo y almacenamiento, fundamentalmente en centros de investigación y universidades. Estos centros comenzaron a ofrecer sus servicios a terceros a través de protocolos estándar, constituyendo la denominada arquitectura de computación grid, orientada al procesamiento en paralelo o al almacenamiento de gran cantidad de información. Estas arquitecturas fueron acogidas en instituciones investigadoras durante la primera mitad de la década de 2000, pero la complejidad para utilizar la infraestructura, las dificultades para utilizar diferentes grids, y los problemas de portabilidad entre ellas, hicieron que nunca se popularizara fuera del ámbito de la investigación y académico. Durante esta misma época comenzaron a popularizarse las tecnologías de virtualización que hacían posible implementar máquinas virtuales que “desacopla” el hardware del software y permiten replicar el entorno del usuario sin tener que instalar y configurar todo el software que requiere cada aplicación. Esto tiene ventajas en la distribución y mantenimiento de sistemas de software complejos y permite integrar bajo un mismo entorno un conjunto de sistemas heterogéneos. Esta nueva arquitectura permitía distribuir carga de trabajo de forma sencilla, lo cual elimina los problemas que presentaba la arquitectura grid, abriendo una nueva puerta al cálculo distribuido, llamado cloud computing. Este nuevo modelo emerge como un nuevo paradigma capaz de proporcionar recursos de cálculo y de almacenamiento que, además, resulta especialmente apto para la explotación comercial de las grandes capacidades de cómputo de proveedores de servicios en Internet.

Tras la evolución descrita en el apartado anterior, el concepto de cloud computing se ha establecido en los últimos años y se consolida como nuevo paradigma de cálculo o escenario de plataformas TI. Atendiendo a la definición dada por el NIST (National Institute of Standards and Technology), el cloud computing es un modelo tecnológico que permite el acceso ubicuo, adaptado y bajo demanda en red a un conjunto compartido de recursos de computación configurables compartidos (por ejemplo: redes, servidores, equipos de almacenamiento, aplicaciones y servicios), que pueden ser rápidamente aprovisionados y liberados con un esfuerzo de gestión reducido o interacción mínima con el proveedor del servicio. Otra definición complementaria es la aportada por el RAD Lab de la Universidad de Berkeley, desde donde se explica que el cloud computing se refiere tanto a las aplicaciones entregadas como servicio a través de Internet, como el hardware y el software de los centros de datos que proporcionan estos servicios. Los servicios anteriores han sido conocidos durante mucho tiempo como Software as a Service (SaaS), mientras que el hardware y software del centro de datos es a lo que se llama nube . Con la información sintetizada en este apartado, se entiende que el cloud computing representa un cambio importante en cómo pueden las empresas y Organismos Públicos procesar la información y gestionar las áreas TIC; apreciándose que con la gestión TIC tradicional las empresas realizan cuantiosas inversiones en recursos, incluyendo hardware, software, centros de procesamiento de datos, redes, personal, seguridad, etc.; mientras que con los modelos de soluciones en la nube se elimina la necesidad de grandes inversiones y costes fijos, transformando a los proveedores en empresas de servicios que ofrecen de forma flexible e instantánea la capacidad de computación bajo demanda.



Motivación

En un mundo tecnológico como el nuestro, y tratándose de la rama estudiada, en constante cambio, es necesario permanecer en continuo aprendizaje. Las tecnologías del Cloud Computing son muy jóvenes en cuanto a años se refiere, pero han gozado de una maduración durante los años, porque la tecnología lo ha permitido. Desembocando, incluso, hacia un nuevo paradigma llamado Cloud Computing. Todo apunta a que se convertirá en algo imprescindible en un futuro. Es por ello, que es necesario conocer las bases que la componen, las tecnologías que lo utilizan y las metodologías inherentes a ellas.

A medida de resumen, es de vital importancia conocer al menos en la medida de lo posible, todas las corrientes de pensamiento que surgen tras el paso del tiempo. Es por esto, que la tipología de la arquitectura estudiada esta práctica, está motivada por este factor. Con la finalidad de ampliar dicho conocimiento sobre las nuevas tecnologías que aparecen.

Objetivos

Se pretende llegar a entender el funcionamiento básico del Cloud Computing, y explotarlo con una aproximación de software propia. Para ello se llevará a cabo un estudio de las diferentes variantes de Cloud Computing, con la mejor aproximación a ellas, o incluso la utilización de distintas soluciones en conjunto. Los lenguajes de programación, que presentan una mejor posición hacia este tipo de ámbitos; y las distintas metodologías que pueden aplicarse para llegar a una solución óptima y eficaz.

Anteproyecto

Javascript

Javascript es un lenguaje de programación que surgió con el objetivo inicial de programar ciertos comportamientos sobre las páginas web, respondiendo a la interacción del usuario y la realización de automatismos sencillos. En ese contexto podríamos decir que nació como un "lenguaje de scripting" del lado del cliente, sin embargo, hoy Javascript es mucho más. Las necesidades de las aplicaciones web modernas y el HTML5 ha provocado que el uso de Javascript que encontramos hoy haya llegado a unos niveles de complejidad y prestaciones tan grandes como otros lenguajes de primer nivel. El lenguaje interpretado, dialecto del estándar ECMAScript. Es un lenguaje de tipo:

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

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

- funcional es un paradigma de programación que se basa en la evaluación de expresiones, como son las funciones, evitando tener estados y datos mutables, a diferencia de la programación imperativa que se basa en cambios de estado a través de la ejecución de instrucciones. JavaScript posee características de la programación funcional como son las Funciones de Orden Superior. Se denominan así a las funciones que admiten por parámetro otras funciones o que como resultado devuelve una función.

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

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

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

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

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

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

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

En los últimos años Javascript se está convirtiendo también en el lenguaje "integrador". Lo encontramos en muchos ámbitos, ya no sólo en Internet y la Web, también es nativo en sistemas operativos para ordenadores y dispositivos, del lado del servidor y del cliente. Tenemos como ejemplos, nodeJS, una implementación de Javascript del lado del servidor, por módulos diseñados en C, que le profieren una gran capacidad para trabajar en términos de redes, a nivel de byte. Por otro lado, se ha convertido en una interesante alternativa, en conjunto de otras tecnologías, como HTML5 y CSS3, para proveer a las aplicaciones de multiplataforma entre diferentes dispositivos. Se han creado aplicaciones que embeben navegadores, con sus diferentes implementaciones en cada uno de los sistemas operativos de distintos dispositivos, IOs, Android, Windows Phone. Gracias a su navegador embebido, la aplicación, con el mismo código, es capaz de funcionar en todos los dispositivos de igual forma. Aquella visión de Javascript aislado a pequeñas funcionalidades, añadir texto a un botón, enviar un alert, se ha quedado muy pequeña. Todas las nuevas tecnologías confluyen hacia el standard de la web. Con ello, las especificaciones de los lenguajes, profieren cada vez, unas características más interesantes, día a día. La Web 2.0 se basa en el uso de Javascript para

implementar aplicaciones enriquecidas que son capaces de realizar todo tipo de efectos, interfaces de usuario y comunicación asíncrona con el servidor por medio de Ajax.

La utilización de este lenguaje, cada día crece más, y surgen nuevos Frameworks. Es el lenguaje que más ha crecido en cuanto a actividad de la comunidad en GitHub. En el periodo de Q2/2012 a Q4/2014, ha pasado de tener 84.893 a 323.938 repositorios, 3,82 veces su inicial balance. Los nuevos frameworks, se focalizan en diferentes ámbitos, y todos ellos tienen un gran soporte de la comunidad. En parte su gran crecimiento, es debido al gran aporte desinteresado de personas que quieren aportar sus conocimientos para que todo siga en contínuo desarrollo.

La aparición de nuevos frameworks, ha encaminado hacia cambios muy positivos al lenguaje, en su estandarización. Técnicas, métodos e ideas, ya han sido probados por la comunidad a través de diferentes librerías. Nuevos términos surgen con el la aparición de librerías, algunos tan conocidos como SPA, de la mano de un gigante de la tecnología, Google. Muchas otras empresas desarrollan sus propias herramientas, Facebook con React.JS, Twitter con Bootstrap y Bower, etc. Su uso es libre y totalmente gratuito, lo que provoca que la comunidad use use las herramientas, apoye el proyecto, y ayude a su mantenimiento y evolución. Así como, se han introducido patrones tan importantes en el mundo del software como, MVC, a través de metodologías, y con la ayuda de diferentes frameworks, que han elevado el nivel del lenguaje, convirtiéndolo en un lenguaje muy bien estructurado, a pesar de su diseño(no fue pensado para estas características).

El motor V8 de Google

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

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

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

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

- compilación y ejecución de código JavaScript: el código fuente se pasa código máquina cuando es cargado y antes de ser ejecutado por primera vez.

- recolección eficiente de basura: este término se utiliza al hablar de la liberación de memoria que ocupan los objetos JavaScript cuando no van a usarse más. V8 emplea un recolector stop-the-world, es decir, V8, cíclicamente, detiene la ejecución del programa (en la llamada “pausa embarazosa”) procesando sólo una parte de la memoria heap para minimizar el efecto de la parada. Durante esta parada no se crean nuevos objetos pero también se evita con ella que no haya indisponibilidad momentánea de los existentes.

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

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

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

NodeJS

Introducción

Proyecto creado por Ryan Dahl a principios de 2009 cuya principal propósito es la creación de aplicaciones para Internet, principalmente Web, propiciado por la tarea que desempeñaba en aquella época. La idea empezó a gestarse a partir de otro proyecto para el framework Ruby on Rails, un pequeño y rápido servidor web llamado Ebb, también de la autoría de Ryan Dahl, que evolucionó a una librería en C. El aspecto negativo, era la complejidad que suponía el lenguaje C para programar aplicaciones. Es en este punto es donde entra en escena el, por aquel entonces, recién aparecido intérprete de JavaScript de Google, V8, que no tardó en convertirse en el motor referencia(puesto en marcha en el navegador Chrome, también propietario de Google).

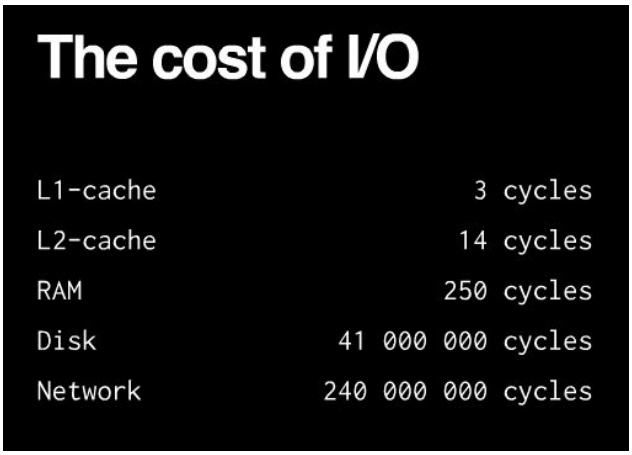
El principal propósito de la evolución del proyecto desde Ruby a C, y posteriormente, de C a JavaScript fue el de realizar un sistema en cual, la Entrada/Salida fuera enteramente no bloqueante. Ryan tenía claro que era esencial para obtener un alto rendimiento. Con Ruby y C siempre había una parte del sistema que era bloqueante. JavaScript, por el contrario, se ajusta a este requisito, debido a la arquitectura en la que está diseñado.

Para ejecutarse en un bucle de eventos, que es, precisamente, lo que Node hace: delegar en la plataforma las operaciones de Entrada/Salida que solicita la aplicación. De esta manera Node puede seguir realizando tareas sin estar bloqueado esperando, y cuando las operaciones se han completado, procesará en su bucle de eventos el evento generado y ejecutará el código que lo maneja según se haya definido. La consecuencia de este modelo Entrada/Salida es que se puede atender a un altísimo número de clientes a la vez, motivo que ha llevado a Node a ser el paradigma de plataforma de aplicaciones de tiempo real.

Desde la presentación de Node, el proyecto no ha parado de crecer. Actualmente, es uno de los repositorios más populares en Github , con más de 37.000 seguidores, y tiene más de 164.509 librerías, a las que llaman módulos, registradas en la web de su gestor de paquetes, NPM . Desde la fecha de su creación ha acumulado un total de más de 8.400 forks. Además de la gran actividad en Internet, a la que hay que sumarle el concurso mundial de aplicaciones Node Knockout , la comunidad de Node tiene una cita anualmente con el ciclo de conferencias NodeConf , donde se presentan todos los avances de la plataforma, o con la JSConf , un evento para desarrolladores de JavaScript donde la plataforma es protagonista de muchas de las conferencias que se realizan. Node está apadrinado por la compañía Joyent , que contrató a Ryan Dahl cuando comenzaba el proyecto. Joyent ofrece, conjuntamente con Nodejistsu , como IaaS, un entorno en “la Nube” donde desplegar aplicaciones Node. Pero no sólo Joyent ofrece alojamiento para esta plataforma. Heroku también tiene soluciones cloud-computing personalizables, y, si se eligen opciones gratuitas y open source, Nodester da espacio para aplicaciones como PasS. Como colofón, Node fue galardonado en 2012 con el premio “Tecnología del Año” por la revista InfoWorld, perteneciente a una división prestigioso grupo internacional de prensa especializada IDG. Y, posiblemente, todo no haya hecho más que empezar.

¿Qué es Node?

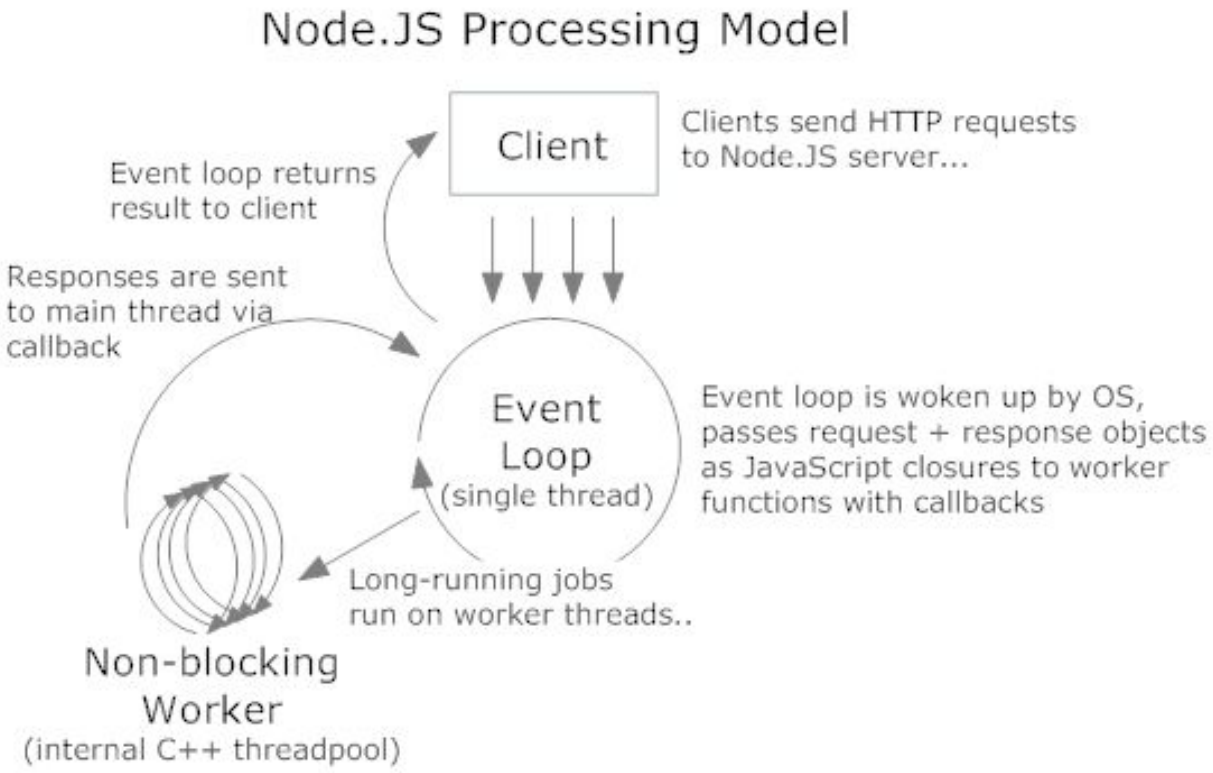
La mejor manera de aproximarse a Node es a través de la definición que aparece en su página web: “Node.js es una plataforma construida encima del entorno de ejecución javascript de Chrome para fácilmente construir rápidas, escalables aplicaciones de red. Node.js usa un modelo de E/S no bloqueante dirigido por eventos que lo hace ligero y eficiente, perfecto para aplicaciones data-intensive en tiempo real” Esta visión global de Node se puede diseccionar en pequeñas partes que, una vez analizadas separadamente, dan una visión mucho más precisa y detallada de las características del proyecto.



“Es una plataforma”

En efecto, Node provee un entorno de ejecución para un determinado lenguaje de programación y un conjunto de librerías básicas, o módulos nativos, a partir de las cuales crear aplicaciones orientadas principalmente a las redes de comunicación, aunque una parte de estas librerías permiten interactuar con componentes del sistema operativo a través de funciones que cumplen con el estándar POSIX. Básicamente este estándar o familia de estándares define las interfaces y el entorno, así como utilidades comunes, que un sistema operativo debe soportar y hacer disponibles para que el código fuente de un programa sea portable (compilable y ejecutable) en diferentes sistemas operativos que implementan dicho estándar. En este caso, Node facilita funciones para manejo de archivos, Entrada/Salida, señales y procesos conformes a las características establecidas por POSIX.

El proceso de arranque de Node



El código que realiza el arranque, o bootstrap, del núcleo de la plataforma es lo primero que se ejecuta y uno de sus cometidos es proveer el mecanismo para cargar el resto de los módulos del core según vayan siendo necesarios o se demanden. Se puede echar un vistazo a este código en el fichero del código fuente src/node.js.

```
function startup() {
    var EventEmitter = NativeModule.require('events').EventEmitter;

    process.__proto__ = Object.create(EventEmitter.prototype, {
      constructor: {
        value: process.constructor
      }
    });
    EventEmitter.call(process);

    process.EventEmitter = EventEmitter; // process.EventEmitter is
deprecated

    // do this good and early, since it handles errors.
    startup.processFatal();

    startup.globalVariables();
    startup.globalTimeouts();
    startup.globalConsole();

    startup.processAssert();
    startup.processConfig();
    startup.processNextTick();
    startup.processStdio();
    startup.processKillAndExit();
    startup.processSignalHandlers();

}
```

Ahí se ve que el encargado de realizar la carga es el objeto NativeModule, que ofrece un minimalista sistema de gestión de módulos. Pero aparte de NativeModule, en el proceso de arranque ocurren muchas más cosas, de las que se encargan el resto de funciones presentes en node.js. Una de las primeras acciones que se realizan es hacer disponibles las variables, objetos y funciones globales. Por globales se entiende que están disponibles en el scope donde corre Node, que se hace disponible al programa mediante, precisamente, la variable global. Por defecto, disponemos de las funciones que ofrecen los módulos console y timers, el objeto Buffer, del módulo buffer, y el objeto nativo process. Se hablará de process por ser quizás uno de los objetos más interesantes de la plataforma desde el punto de vista del diseño. Presente en el espacio global de ejecución del proceso principal de Node representa a ese mismo proceso. Como se ha dicho, se hace disponible en él después de crearse en código nativo a través de la función SetupProcessObject() en el proceso de arranque definido en el fichero src/node.cc10. Con esta función se crea un objeto al que se le añaden propiedades y métodos en código nativo que luego están disponibles para el programador a través del API. Éstos permiten:

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

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

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

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

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

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

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

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

- Se establece la clase EventEmitter su como prototipo, con lo cual proccess hereda sus métodos y por tanto la capacidad de generar eventos y notificarlo a sus subscriptores. Se profundizará en EventEmitter cuando se hable de la arquitectura de Node.
- Se establecen los métodos de interacción con el bucle de eventos 11. El bucle de eventos es una característica de la arquitectura de Node que tiene mucho que ver con el altísimo rendimiento de la plataforma. Más adelante se analizará en detalle este punto. A nivel de programador, el API ofrece process.nextTick().

- Se inicializan los Streams de Entrada/Salida estándar 12. El API los presenta como las propiedades: process.stdout, process.stderr, habilitados ambos para la escritura y, a diferencia de los demás Streams de Node, son bloqueantes; y process.stdin que es de lectura y está pausado de inicio cuando la entrada estándar no es otro Stream, con lo que, para leer de él, se debe abrir primero invocando a process.openStdin().
- Se definen los métodos relacionados con las señales13 del sistema operativo: process.exit(), para terminar la ejecución del programa especificando un código de salida, y process.kill(), para enviar señales a otros procesos, tal y como se haría con la llamada al sistema operativo kill.
- Se añade funcionalidad a los métodos para manejar listeners14 que hereda de EventEmitter, recubriéndolos para que sean capaces de escuchar y actuar sobre señales del sistema operativo (tipo Unix), no sólo eventos.
- Se determina si, en lugar de ser el proceso principal de Node, el programa es un proceso worker del modulo cluster, de los que se hablará cuando se comente el entorno de ejecución de Node.
-

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

Script

Se accede a este modo pasando por línea de comandos el nombre de un fichero .js que contiene el código del programa. Node cargará ese fichero y lo ejecutará. Es el modo más común de trabajar en la plataforma. También se puede indicar por línea de comandos mediante el modificador -e
o –eval seguido del nombre del script.

REPL

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

Debug

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

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

El objeto que carga las librerías, como se ha comentado, es NativeModule el cual ofrece un pequeño mecanismo para la carga y gestión de módulos. Este mecanismo es auxiliar y una vez cumple su función se reemplaza por la funcionalidad, más completa, que ofrece el módulo module, cargado a su vez, paradójicamente, por NativeModule. Cuando se desea tener disponible cualquier módulo para usarlo en un programa, el método require() que se invoca es el de module15 . Este método comprueba, con ayuda de NativeModule, si el módulo está en la caché de módulos y en caso negativo, lo busca en el sistema de ficheros o en caso de que sea un módulo del core de Node, en el propio ejecutable de la plataforma, ya que éstos están embebidos en él [6]. Una vez localizado, lo compila, lo mete en la caché y devuelve su exports, es decir, las funciones, variables u objetos que el módulo pone a disposición del programador.

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

El formato CommonJS

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

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

Las implicaciones de este estándar son:

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

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

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

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

Módulos disponibles en el core de Node

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

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

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

module
Proporciona el sistema de módulos según impone CommonJS. Cada módulo que se carga o el propio programa, está modelado según module, que se verá como una variable, module, dentro del mismo módulo. Con ella se tienen disponibles tanto el mecanismo de carga require() como aquellas funciones y variables que exporta, en module.exports, que destacan entre otras menos corrientes que están a un nivel informativo: módulo que ha cargado el actual (module.parent), módulos que carga el actual (module.children)...

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

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

util
Conjunto de utilidades principalmente para saber si un objeto es de tipo array, error, fecha, expresión regular...También ofrece un mecanismo para extender clases de JavaScript a través de herencia: inherits(constructor, superConstructor);

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

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

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

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

string_decoder
Proporciona una manera de, a partir de un Buffer, obtener cadenas de caracteres codificados en utf-8.

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

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

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

dgram
Creación y manejo asíncrono de datagramas sobre el protocolo transporte UDP.

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

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

https
Versión del protocolo HTTP sobre conexiones seguras TLS/SSL.

url
Formateo y análisis de los campos de las URL.

querystrings
Utilidades para trabajar con las queries en el protocolo HTTP. Una query son los parámetros que se envían al servidor en las peticiones HTTP. Dependiendo del tipo de petición (GET o POST), pueden formar parte de la URL por lo 17 que deben codificarse o escaparse y concatenarse de una manera especial para que sean interpretadas como tal.

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

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

vm
Compilación y ejecución bajo demanda de código.

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

assert
funciones para la escritura de tests unitarios.

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

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

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

_debugger
Es el depurador de código que Node tiene incorporado, a través de la opción debug de la línea de comandos. En realidad es un cliente que hace uso 18 de las facilidades de depuración que el intérprete de Javascript que utiliza Node ofrece a través de una conexión TCP al puerto 5858. Por tanto, no es un módulo que se importe a través de require() sino el modo de ejecución Debug del que se ha hablado antes.

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

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

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

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

_linklist
Implementa una lista doblemente enlazada. Esta estructura de datos se emplea en timers.js, el módulo que provee funcionalidad de temporización. Su función es encadenar temporizadores que tengan el mismo tiempo de espera, timeout. Esta es una manera muy eficiente de manejar enormes cantidades de temporizadores que se activan por inactividad, como los timeouts de los sockets, en los que se reinicia el contador si se detecta actividad en 19 él. Cuando esto ocurre, el temporizador, que está situado en la cabeza de la lista, se pone a la cola y se recalcula el tiempo en que debe expirar el primero.

buffer_ieee754
Implementa la lectura y escritura de números en formato de coma flotante según el estándar IEEE754 del IEEE16 que el módulo buffer emplea para las operaciones con Doubles y Floats.

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

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

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

Comunidad Open Source

Cita de http://www.gnu.org/philosophy/open-source-misses-the-point.html

Por qué el «código abierto» pierde de vista lo esencial del software libre

por **Richard Stallman**

Cuando decimos que el software es «libre», nos referimos a que respeta las libertades esenciales del usuario: la libertad de utilizarlo, ejecutarlo, estudiarlo y modificarlo, y de distribuir copias con o sin modificaciones. Es una cuestión de libertad y no de precio, por lo tanto piense en «libertad de expresión» y no en «barra libre». [1]
Estas libertades son de vital importancia. Son esenciales no solamente para el bien del usuario individual sino para la sociedad entera, porque promueven la solidaridad social: compartir y cooperar. La importancia de estas libertades aumenta a medida que nuestra cultura y nuestras actividades cotidianas se vinculan cada vez más con el mundo digital. En un mundo de sonidos, imágenes y palabras digitales, el software libre se vuelve cada vez más esencial para la libertad en general.
Decenas de millones de personas alrededor del mundo ahora utilizan software libre; las escuelas públicas de algunas regiones de India y España enseñan a todos los estudiantes a utilizar el sistema operativo libre GNU/Linux. Sin embargo, la mayoría de estos usuarios nunca han oído las razones éticas por las cuales desarrollamos este sistema y construimos la comunidad del software libre, porque este sistema y esta comunidad son descritos como «de código abierto» y atribuidos a una filosofía diferente que rara vez menciona estas libertades.

El movimiento del software libre ha hecho campaña por la libertad de los usuarios de ordenador desde 1983. En 1984 iniciamos el desarrollo del sistema operativo libre GNU, para poder evitar el uso de sistemas operativos que no son libres y que niegan la libertad a los usuarios. Durante los años ochenta desarrollamos la mayor parte de los componentes esenciales del sistema GNU, y diseñamos la Licencia Pública General de GNU (GNU GPL, por sus siglas en inglés) para usarla en la distribución de dichos componentes; una licencia diseñada específicamente para proteger la libertad de todos los usuarios de un programa.
Sin embargo, no todos los usuarios y programadores de software libre estaban de acuerdo con los objetivos del movimiento del software libre. En 1998 una parte de la comunidad del software libre se bifurcó y dió inicio a una campaña para promover el «open source» (código abierto). La expresión se propuso originalmente para evitar un posible malentendido con el término «free software» (software libre), pero pronto se asoció con posiciones filosóficas diferentes a las del movimiento del software libre.

Algunos de los defensores del «código abierto» lo consideraron una «campaña de marketing para el software libre», con el objetivo de atraer a los ejecutivos de las empresas enfatizando los beneficios prácticos sin mencionar conceptos de lo que es correcto e incorrecto —y que quizá los empresarios no deseaban oír—. Otros defensores rechazaban rotundamente los valores éticos y sociales del software libre. Cualesquiera que hayan sido sus puntos de vista, cuando hacían campaña por el «código abierto» no mencionaban ni abogaban por esos valores. La expresión «código abierto» fue rápidamente asociada con ideas y argumentaciones basadas únicamente en valores de orden práctico, tales como desarrollar o usar software potente y confiable. La mayoría de los partidarios del «código abierto» llegaron al movimiento después de entonces y hacen la misma asociación de conceptos.

Ambas expresiones describen casi la misma categoría de software, pero representan puntos de vista basados en valores fundamentalmente diferentes. El código abierto es una metodología de programación, el software libre es un movimiento social. Para el movimiento del software libre, el software libre es un imperativo ético, respeto esencial por la libertad de los usuarios. En cambio la filosofía del código abierto plantea las cuestiones en términos de cómo «mejorar» el software, en sentido meramente práctico. Sostiene que el software privativo no es una solución óptima para los problemas prácticos que hay que resolver. En la mayoría de los casos, cuando se discute sobre «código abierto» no se toma en consideración el bien y el mal sino únicamente la popularidad y el éxito; he aquí un ejemplo típico.

Para el movimiento del software libre, sin embargo, el software que no es libre es un problema social y la solución consiste en dejar de usarlo, migrar al software libre.

«Software libre». «Código abierto».

Si es el mismo software (o casi), ¿importa acaso qué nombre se utiliza? Sí, porque las diferentes palabras expresan ideas diferentes. Aunque un programa libre con cualquier otro nombre le dará hoy la misma libertad, establecer la libertad de manera perdurable depende sobre todo de enseñar a las personas a valorar la libertad. Si desea ayudar en esto, es esencial que use la expresión «software libre».

Nosotros, en el movimiento del software libre, no vemos el ámbito del código abierto como al enemigo; el enemigo es el software privativo, el que no es libre. Pero queremos que la gente sepa que defendemos la libertad, así que no aceptamos que se nos identifique como partidarios del código abierto.

Diferencias prácticas entre software libre y código abierto

En la práctica, el código abierto sostiene criterios un poco más débiles que los del software libre. Por lo que sabemos, todo el software libre existente se puede calificar como código abierto. Casi todo el software de código abierto es software libre, con algunas excepciones. En primer lugar, algunas licencias de código abierto son demasiado restrictivas, por lo que no se las puede considerar licencias de software libre. Por ejemplo, «Open Watcom» no es libre, ya que su licencia no permite hacer versiones modificadas y utilizarlas de forma privada. Afortunadamente, son muy pocos los programas que llevan tales licencias.

En segundo lugar, y lo que en la práctica es más importante, muchos productos que funcionan como ordenadores verifican las firmas de sus programas ejecutables para impedir que los usuarios instalen ejecutables diferentes; solo una compañía tiene el privilegio de elaborar ejecutables que funcionen en el dispositivo y de acceder a todas las prestaciones del mismo. A estos dispositivos los llamamos «tiranos» y la práctica se denomina «tivoización», por referencia al producto (Tivo) donde por primera vez descubrimos su implementación. Aun cuando el ejecutable esté hecho a partir de código fuente libre, los usuarios no pueden ejecutar versiones modificadas, de modo que el ejecutable no es libre.

Los criterios del código abierto no contemplan esta cuestión, solo les interesa la licencia del código fuente. De modo que estos ejecutables no modificables, si están hechos a partir de un código fuente como Linux, que es de código abierto y libre, son de código abierto pero no son libres. Muchos productos Android contienen ejecutables tivoizados de Linux, que no son libres.

Errores frecuentes sobre el significado de «software libre» y «código abierto»

La expresión «software libre» puede dar lugar a un malentendido. El significado no intencional —«software que se puede obtener sin costo alguno»— corresponde, y también corresponde el significado con el que lo usamos: «software que da al usuario ciertas libertades». Resolvemos este problema publicando la definición de software libre, donde decimos: «Piense en libertad de expresión, no en barra libre». No es una solución perfecta, no puede eliminar completamente el problema. Un término correcto e inequívoco sería mejor, si no presentase otros problemas.

Lamentablemente, todas las alternativas en inglés presentan algún problema. Hemos analizado muchas alternativas que nos han sugerido, pero ninguna es tan claramente «correcta» como para adoptarla. Por ejemplo, en ciertos contextos se puede usar la palabra española y francesa «libre», pero en India no la reconocerán en absoluto. Todos las alternativas propuestas para «software libre» tienen algún tipo de problema semántico, incluso «software de código abierto».

La definición oficial de «software de código abierto» (publicada por la Open Source Initiative y demasiado larga como para citarla aquí) se derivó indirectamente de nuestros criterios para el software libre. No es la misma, es un poco más amplia en algunos aspectos. No obstante, dicha definición concuerda con la nuestra en la mayoría de los casos.

Sin embargo, el significado obvio de la expresión «software de código abierto» es «puede mirar el código fuente», y pareciera que muchos opinan que eso es lo que significa. Ese es un criterio mucho más débil que la definición de software libre, y también mucho más débil que la definición oficial de código abierto. Incluye muchos programas que no son ni libres ni de código abierto.

Debido a que el significado obvio de «código abierto» no es el que sus defensores quieren darle, el resultado es que la mayoría interpreta erróneamente la expresión. Según el escritor Neal Stephenson, «Linux es software de “código abierto”, lo que significa simplemente que cualquiera puede obtener copias de los archivos del código fuente». No pienso que su intención haya sido rechazar o cuestionar deliberadamente la definición «oficial». Pienso que simplemente aplicó las convenciones del idioma inglés para obtener el significado de la expresión. El estado de Kansas publicó una definición similar: «Utilicen software de código abierto (OSS). OSS es el software cuyo código fuente está disponible pública y libremente, aunque los términos de licenciamento pueden variar con respecto a lo que se puede hacer con el código».

El New York Times publicó un artículo que amplía el significado del término para referirse a las pruebas de los programas beta por parte de los usuarios (se permite a unos cuantos usuarios probar una versión inicial para que den sus impresiones de forma conficendial), algo que los programadores de software privativo han hecho durante décadas.

Los partidarios del código abierto intentan afrontar este problema refiriéndose a su definición oficial, pero ese enfoque correctivo es menos efectivo para ellos que para nosotros. El término «software libre» tiene dos significados naturales, uno de los cuales es el que le damos, de manera que una persona que ha comprendido la idea de «libertad de expresión, no barra libre» no se equivocará de nuevo. Al contrario, el «código abierto» tiene solamente un significado natural, el cual es diferente del que sus partidarios desean darle. Así, no hay una manera concisa de explicar y justificar la definición oficial de «código abierto», lo que causa aún más confusión.

Otra mala interpretación de «código abierto» es la idea de que significa «no usar la GPL de GNU». Esto tiende a provocar otro malentendido: «software libre» equivale a «software que está bajo la GPL de GNU». Ambas interpretaciones son incorrectas, ya que la GPL de GNU se califica como licencia de código abierto, y la mayoría de las licencias de código abierto se consideran licencias de software libre. Existen muchas licencias de software libre además de la GPL de GNU.

El término «código abierto» se ha extendido aún más debido a su aplicación en otras actividades tales como el gobierno, la educación y la ciencia, todos campos en los que no existe nada parecido al código fuente, y donde los criterios aplicables a las licencias de software no son pertinentes. El único elemento que estas actividades tienen en común es que, de algún modo, se invita a las personas a participar. Fuerzan tanto el término que llega a significar únicamente «participación» o «transparencia», o aún menos que eso. En el peor de los casos, se ha convertido en una trivial expresión de moda.

Valores diferentes pueden llevar a conclusiones similares, pero no siempre

Los grupos radicales de los años sesenta tenían la reputación de estar divididos en facciones: algunas organizaciones se apartaban debido a desacuerdos sobre detalles de estrategia, y los dos grupos resultantes se consideraban enemigos aunque tuvieran metas y valores básicos similares. El ala derecha de la política se aprovechó de esto y lo utilizó para criticar a la izquierda en general.

Algunos intentan desacreditar el movimiento de software libre poniendo nuestro desacuerdo con el código abierto en el mismo plano que los desacuerdos entre aquellos grupos radicales. Lo entienden al revés. Estamos en desacuerdo con el código abierto en lo que respecta a los objetivos y valores básicos, pero su perspectiva y la nuestra conducen en muchos casos al mismo comportamiento práctico, por ejemplo, programar software libre.

Como resultado, personas del movimiento del software libre y del ámbito del código abierto a menudo trabajan conjuntamente en proyectos prácticos tales como el desarrollo de software. Es notable que posiciones filosóficas tan diferentes puedan tan a menudo motivar a diferentes personas a participar en los mismos proyectos. Sin embargo, hay situaciones en las que estos puntos de vista fundamentalmente distintos dan como resultado acciones totalmente diferentes. La idea del código abierto es que permitiendo que los usuarios modifiquen y redistribuyan el software se obtienen programas más potentes y confiables. Pero no hay ninguna garantía de que esto sea así. Los programadores de software privativo no son necesariamente incompetentes. A veces producen algún programa potente y confiable, aunque no respete la libertad de los usuarios. La reacción de los activistas del software libre y de los entusiastas del código abierto frente a esa situación será muy diferente.

Un entusiasta puro del código abierto, uno que no esté influenciado para nada por los ideales del software libre, dirá: «Estoy sorprendido de que haya logrado que su programa funcione tan bien sin haber utilizado nuestro modelo de desarrollo, pero lo logró. ¿Cómo puedo obtener una copia?» Esta actitud premia los esquemas que nos quitan la libertad, llevándonos a perderla.
El activista del software libre dirá: «Su programa es muy atractivo pero valoro más mi libertad. Así que rechazo su programa. Haré mi trabajo de alguna otra manera y apoyaré un proyecto para el desarrollo de un reemplazo libre». Si valoramos nuestra libertad, podemos actuar para mantenerla y defenderla.

El software potente y confiable puede ser malo.

El deseo de que el software sea potente y confiable deriva de suponer que tiene que estar diseñado para que resulte útil a sus usuarios. Si es potente y confiable, el software será de mayor utilidad.

Pero sólo se puede decir que el software es útil si respeta la libertad de los usuarios. ¿Qué pasa si el software está diseñado para encadenar a los usuarios? En ese caso la potencia hace que las cadenas sean más restrictivas, y la confiabilidad significa que son más difíciles de quitar. Las funcionalidades maliciosas, como espiar a los usuarios, restringir a los usuarios, las puertas traseras y las actualizaciones impuestas, son comunes en el software privativo, y algunos defensores del código abierto se proponen hacer lo mismo en programas de código abierto.

Bajo la presión de las compañías discográficas y cinematográficas, el software que se pone a disposición de los usuarios está diseñado cada vez más específicamente para restringirlos. Esta funcionalidad maliciosa se conoce como «gestión digital de restricciones» DRM (véase DefectiveByDesign.org) y es la antítesis, en espíritu, de la libertad que el software libre busca proveer. Y no sólo en espíritu, puesto que el objetivo del DRM es pisotear su libertad: los programadores de DRM intentan dificultarle, hacer que le resulte imposible o incluso ilegal modificar los programas que implementan el DRM.

Con todo, algunos partidarios del código abierto han propuesto software con «DRM de código abierto». La idea es que publicando el código fuente de los programas diseñados para restringir su acceso a los datos cifrados, y permitiendo que otros lo modifiquen, se obtendrá software más potente y confiable para restringir a los usuarios como usted. Luego el software se le entregará a usted en dispositivos que no le permitirán modificarlo.

Aunque este software sea de código abierto y utilice el modelo de desarrollo del código abierto, no será software libre ya que no respetará la libertad de los usuarios que en la práctica lo ejecutan. Si el modelo de desarrollo del código abierto logra que este software sea más poderoso y confiable para restringirle a usted como usuario, eso lo hará aún peor.

Miedo de la libertad

Al inicio, la principal motivación de los que decidieron separar el código abierto del ámbito del software libre fue que los planteamientos éticos del «software libre» incomodaban a muchas personas. Es cierto: plantear cuestiones éticas como el tema de la libertad, hablar de responsabilidades y de conveniencia, es inducir a las personas a que se cuestionen cosas que quizá prefieran ignorar, por ejemplo preguntarse si su conducta es ética. Esto puede generar malestar y algunos pueden optar simplemente por ignorar estas cuestiones. Pero esto no quiere decir que tengamos que dejar de hablar de ello.

Sin embargo, eso es lo que decidieron hacer los líderes del «código abierto». Pensaron que omitiendo hablar de ética y de libertad, mencionando únicamente los beneficios prácticos inmediatos de cierto tipo de software libre, podrían «vender» el software más fácilmente a ciertos usuarios, especialmente a las empresas.

Tal enfoque se ha demostrado eficiente, en sus propios términos. La retórica del código abierto ha convencido a muchas empresas y particulares a usar e incluso desarrollar software libre, lo cual ha extendido nuestra comunidad, pero solamente a un nivel práctico y superficial. La filosofía del código abierto, con sus valores puramente prácticos, impide la comprensión de las ideas más profundas del software libre. Trae muchas personas a nuestra comunidad, pero no les enseña cómo defenderla. Eso es bueno, hasta cierto punto, pero no asegura la libertad. Atraer usuarios al software libre los lleva sólo hasta una parte del camino que hay que recorrer para convertirse en defensores de su propia libertad.

Tarde o temprano estos usuarios se sentirán tentados a volver al software privativo por alguna ventaja práctica. Son innumerables las compañías que buscan ofrecer esa tentación, algunas hasta ofrecen copias gratuitas. ¿Qué motivaría a los usuarios a rechazar esto? Sólo si han aprendido a valorar la libertad que el software libre les brinda, a valorar la libertad como tal en vez de la conveniencia técnica y práctica de algún software libre en particular. Para diseminar esta idea, tenemos que hablar acerca de la libertad. Cierta dosis de «silencio» en el trato con las empresas puede resultar útil para la comunidad, pero es peligroso cuando se vuelve tan común que el amor a la libertad llega a verse como una excentricidad.

Esa peligrosa situación es exactamente la que tenemos. Mucha gente relacionada con el software libre, especialmente los que lo distribuyen, habla poco acerca de la libertad; normalmente porque buscan ser «más atractivos para las empresas». Casi todas las distribuciones del sistema operativo GNU/Linux añaden paquetes privativos al sistema libre de base, y con ello invitan a los usuarios a considerar esto como una ventaja en lugar de un defecto.

Las extensiones de software privativas y las distribuciones de GNU/Linux que son parcialmente libres encuentran terrenos fértiles, porque gran parte de nuestra comunidad no insiste en la libertad del software. Esto no es una coincidencia. La mayor parte de los usuarios de GNU/Linux llegaron al sistema por el discurso del «código abierto», el cual no menciona la libertad como una meta. Las prácticas que no sostienen la libertad y las palabras que no hablan de libertad van de la mano, promoviéndose entre sí. Para superar esta tendencia tenemos que hablar más de libertad, y no menos.

Fin de la cita

La comunidad OpenSource, nos brinda un gran cúmulo de ventajas. Podemos obtener herramientas y productos de gran calidad a un coste 0. Es aconsejable ser devolver a la comunidad lo mismo que se recibe de ella. Muchas empresas soportan los costes de mantenimiento de los proyectos puestos en marcha por ellos mismos. A través de portales como GitHub podemos dar nuestra opinión, comunicar errores en el código, mandar mejoras o cambios que podemos desear que incluyan, e incluso, crear nuestro propio repositorio a partir del suyo, para implementar nosotros mismos nuestros propios cambios.

Github se ha convertido en el portal de difusión de código abierto por excelencia. Provee de muchas ventajas y herramientas que le dan la importancia que de la que ahora mismo goza. Podemos proveer a los mantenedores del proyecto de información sobre su código a través de un apartado llamado Issues(incidencias). Podremos crear nuevas conversaciones, así como hablar en anteriores ya creadas por la comunidad. Con la posibilidad de darle etiquetas para marcar de una mejor forma nuestra duda o problema. Además podremos cerrar la incidencia, para indicar a los usuarios que se ha resuelto el problema/duda, o dejarla abierta para que los demás puedan responder abiertamente.

En cuanto a la mejora del código, Github dispone de una característica llama Pull Request, abreviado por la comunidad como PRs. Podemos enviar los cambios a los administradores del repositorio, para que los revisen y en el caso de que quieran integrarlos, que lo hagan con una gran facilidad, ya que la misma plataforma, pone a disposición una herramienta para resolver los conflictos que hayan podidos suceder tras aplicar los commits relacionados con ese PR.

Por último, y como medida más extrema, podemos “clonar” el repositorio a nuestra cuenta de github, para hacer por nosotros mismos los cambios que deseemos. Gracias a la potencia de git, creación de etiquetas, creación de ramas, commits comentados y firmados, podemos solicitar un estado del código muy exacto en el cual podemos incluir nuestras propias mejoras o modificar el código que deseemos.

Es una ventaja a tener en cuenta la búsqueda y utilización de código alojado en portales del tipo de Github. Hay personas muy sabias, que ponen todos sus conocimientos al alcance de todos. La comunidad entera, está continuamente evolucionando la librería, ya sea en mejoras, como en errores o bugs que puedan haber surgido. A nivel de empresa, supone una gran disminución de los costes, partir de algo que ya está hecho, y no solo lo está, sino que tiene mecanismos entre la comunidad para verificar el buen trabajo. Una buena parte de ello, es la documentación que aportar los desarrolladores de la herramienta, así como las preguntas que la gente hace de ella, hacer ver que es algo actual y ampliamente utilizado. Pero no menos importante que la herramienta disponga de archivos llamados test. Existen plataformas de integración contínua que están permanentemente ejecutando los test para probar la buena calidad del código.

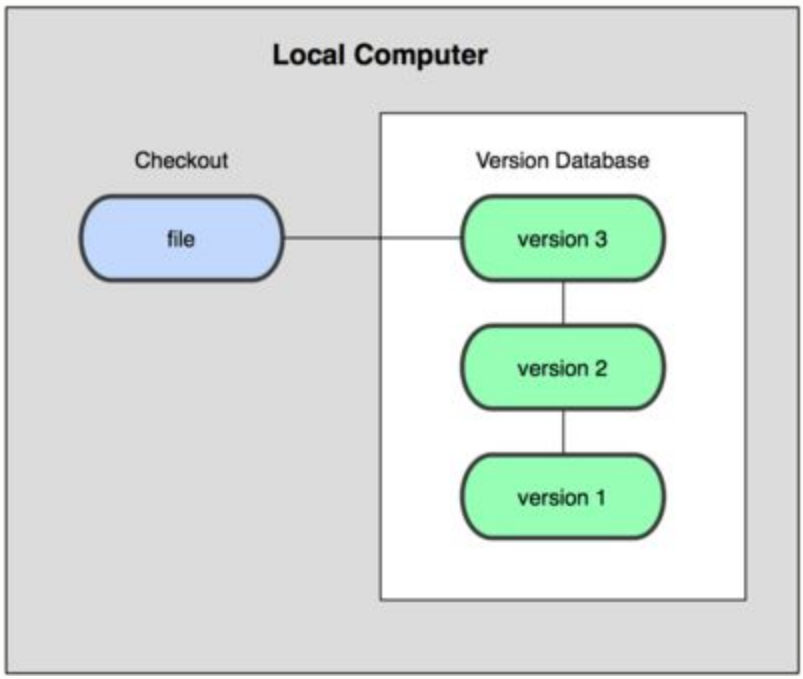
Sistemas de control de versiones

El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas para su posterior uso. Este tipo de sistemas no sólo son aplicables a archivos fuente(en lenguajes de programación), sino también, cualquier tipo de archivo que deseemos llevar un riguroso control, como por ejemplo, las páginas de un libro, configuraciones almacenadas en ficheros de texto o archivos generados por software de terceros.

Sin duda, el sector de la información ha tomado como práctica habitual el uso de este tipo de tecnologías, por su gran versatilidad y gran control de las versiones. Para un diseñador/desarrollador, es un factor importante, tener reflejados todos y cada uno de los cambios, así como su naturaleza. La posibilidad de revertir cambios, permite a los desarrolladores, tener un gran control sobre las distintas versiones, así como prevenir futuras fuentes de errores. También permite la posibilidad de revertir incluso la eliminación de archivos, todo ello a un bajo coste, y con una infraestructura organizada y limpia, que permite la fluidez del código, y el entendimiento entre el equipo de trabajo.

Un método de control de versiones usado por mucha gente es copiar los archivos a otro directorio (quizás indicando la fecha y hora en que lo hicieron, si son avispados). Este enfoque es muy común porque es muy simple, pero también tremendamente propenso a errores. Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.

Para hacer frente a este problema, los programadores desarrollaron hace tiempo VCSs locales que contenían una simple base de datos en la que se llevaba registro de todos los cambios realizados sobre los archivos.



Sistema local.

Una de las herramientas de control de versiones más popular fue un sistema llamado rcs, que todavía podemos encontrar en muchos de los ordenadores actuales. Hasta el famoso sistema operativo Mac OS X incluye el comando rcs cuando instalas las herramientas de desarrollo. Esta herramienta funciona básicamente guardando conjuntos de parches (es decir, las diferencias entre archivos) de una versión a otra en un formato especial en disco; puede entonces recrear cómo era un archivo en cualquier momento sumando los distintos parches.

Sistemas de control de versiones centralizados

El siguiente gran problema que se encuentra la gente es que necesitan colaborar con desarrolladores en otros sistemas. Para solventar este problema, se desarrollaron los sistemas de control de versiones centralizados (Centralized Version Control Systems o CVCSs en inglés). Estos sistemas, como CVS, Subversion, y Perforce, tienen un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central. Durante muchos años éste ha sido el estándar para el control de versiones.

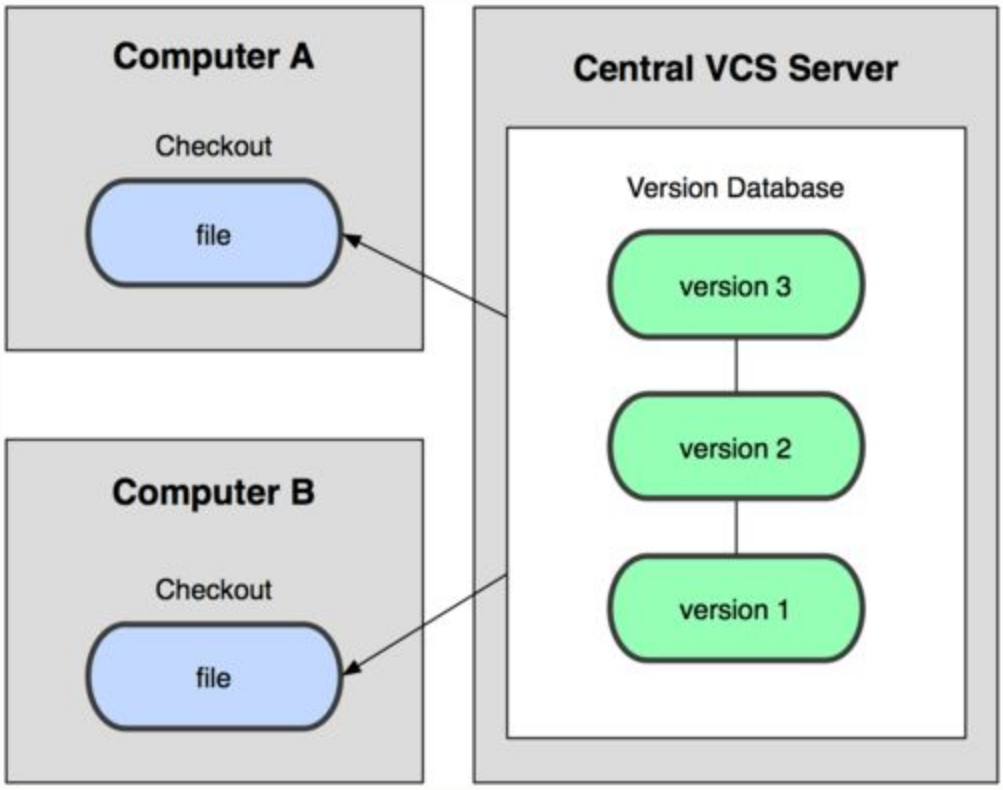


Diagrama de control de versiones centralizado.

Esta configuración ofrece muchas ventajas, especialmente frente a VCSs locales. Por ejemplo, todo el mundo puede saber (hasta cierto punto) en qué están trabajando los otros colaboradores del proyecto. Los administradores tienen control detallado de qué puede hacer cada uno; y es mucho más fácil administrar un CVCS que tener que lidiar con bases de datos locales en cada cliente.

Sin embargo, esta configuración también tiene serias desventajas. La más obvia es el punto único de fallo que representa el servidor centralizado. Si ese servidor se cae durante una hora, entonces durante esa hora nadie puede colaborar o guardar cambios versionados de aquello en que están trabajando. Si el disco duro en el que se encuentra la base de datos central se corrompe, y no se han llevado copias de seguridad adecuadamente, pierdes absolutamente todo.

Sistemas de control de versiones distribuidos

Es aquí donde entran los sistemas de control de versiones distribuidos (Distributed Version Control Systems o DVCSs en inglés). En un DVCS (como Git, Mercurial, Bazaar o Darcs), los clientes no sólo descargan la última instantánea de los archivos: replican completamente el repositorio. Así, si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo. Cada vez que se descarga una instantánea, en realidad se hace una copia de seguridad completa de todos los datos.

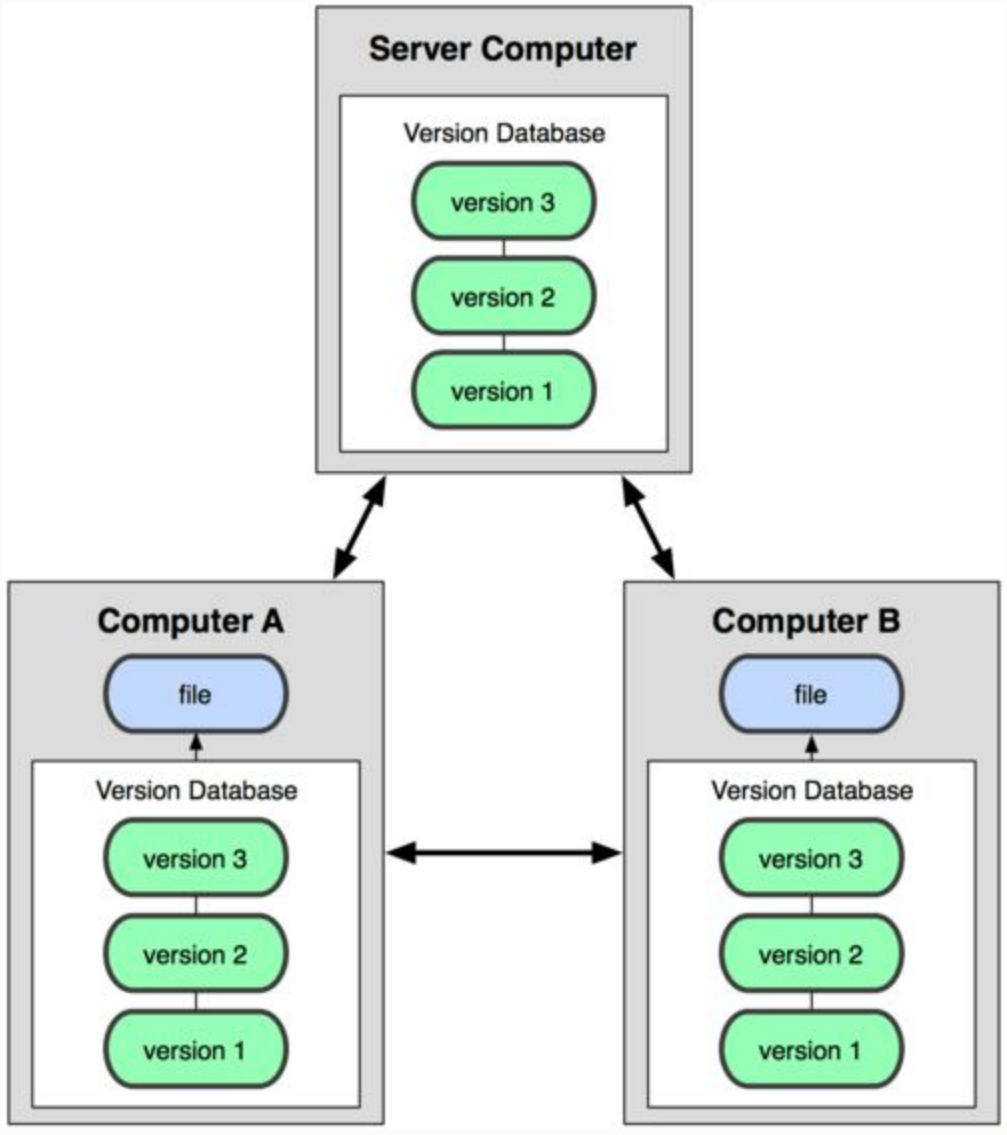


Diagrama de control de versiones distribuido.

Git

Git comenzó con un poco de destrucción creativa y encendida polémica. El núcleo de Linux es un proyecto de software de código abierto con un alcance bastante grande. Durante la mayor parte del mantenimiento del núcleo de Linux (1991-2002), los cambios en el software se pasaron en forma de parches y archivos. En 2002, el proyecto del núcleo de Linux empezó a usar un DVCS propietario llamado BitKeeper.

En 2005, la relación entre la comunidad que desarrolla el núcleo de Linux y la compañía que desarrollaba BitKeeper se vino abajo, y la herramienta dejó de ser ofrecida gratuitamente. Esto impulsó a la comunidad de desarrollo de Linux (y en particular a Linus Torvalds, el creador de Linux) a desarrollar su propia herramienta basada en algunas de las lecciones que aprendieron durante el uso de BitKeeper. Algunos de los objetivos del nuevo sistema fueron los siguientes:

- Velocidad
- Diseño sencillo
- Fuerte apoyo al desarrollo no lineal (miles de ramas paralelas)
- Completamente distribuido
- Capaz de manejar grandes proyectos (como el núcleo de Linux) de manera eficiente (velocidad y tamaño de los datos)

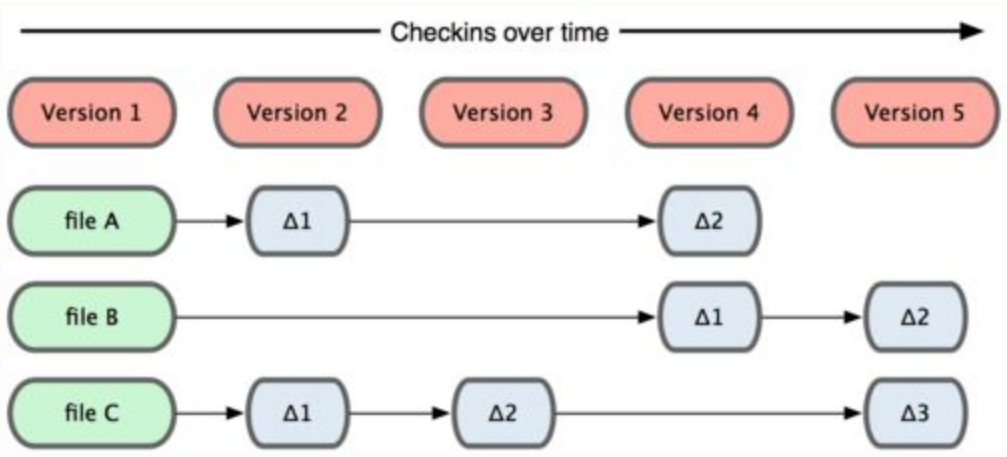
Desde su nacimiento en 2005, Git ha evolucionado y madurado para ser fácil de usar y aún conservar estas cualidades iniciales. Es tremendamente rápido, muy eficiente con grandes proyectos, y tiene un increíble sistema de ramificación (branching) para desarrollo no lineal.

Fundamentos de Git

Git almacena y modela la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz sea bastante similar.

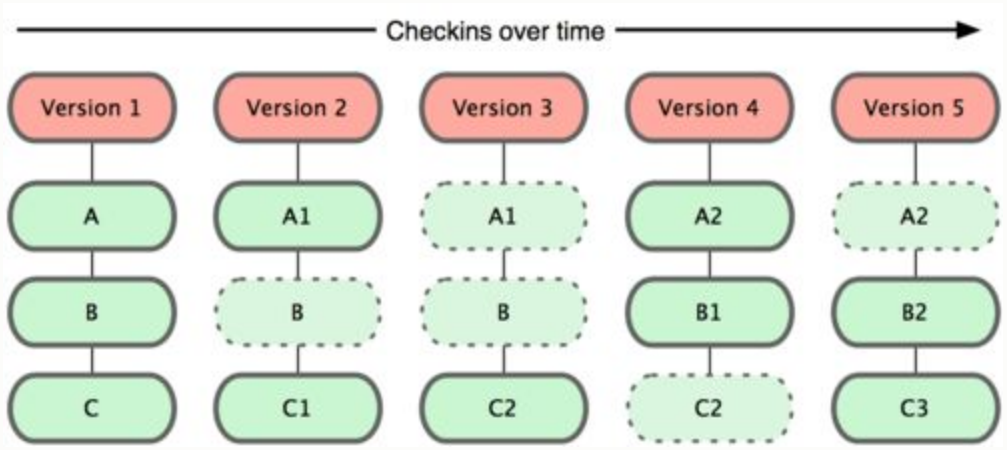
Instantáneas, no diferencias

La principal diferencia entre Git y cualquier otro VCS es cómo Git modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, como ilustra la siguiente figura.



Otros sistemas tienden a almacenar los datos como cambios de cada archivo respecto a una versión base.

Git no modela ni almacena sus datos de este modo. En cambio, Git modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente hace una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado. Git modela sus datos más como en la siguiente figura.



Git almacena la información como instantáneas del proyecto a lo largo del tiempo.

Esta es una distinción importante entre Git y prácticamente todos los demás VCSs. Hace que Git reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que Git se parezca más a un sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, que a un VCS.

La gran totalidad de operaciones es en ámbito local

La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de tu red. Cómo tienes toda la historia del proyecto ahí mismo, en tu disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Git tiene integridad

Todo en Git es verificado mediante una suma de comprobación (checksum) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No es posible la pérdida de información durante su transmisión o sufrir corrupción de archivos sin que Git lo detecte.
El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales, y se calcula en base a los contenidos del archivo o estructura de directorios.Un hash realizado con este algoritmo, tiene una apariencia similar a:

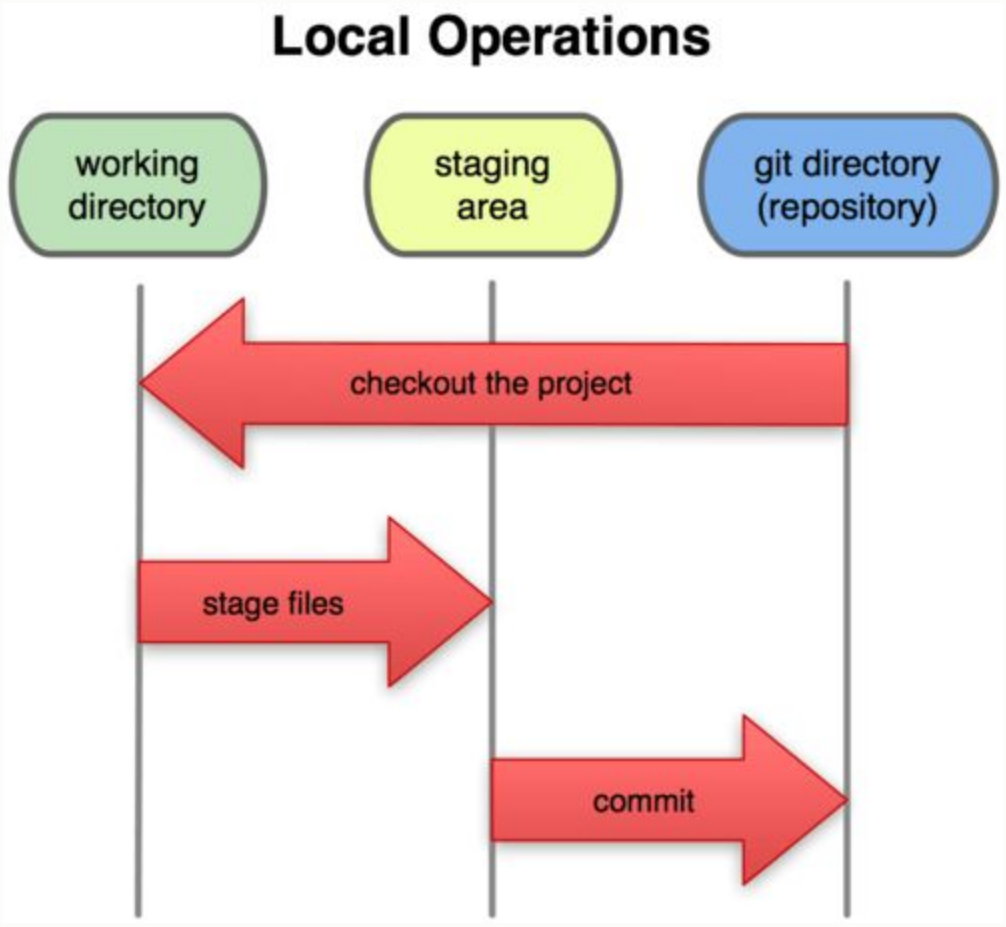
24b9da6552252987aa493b52f8696cd6d3b00373

Git generalmente sólo añade información

Con la ejecución de acciones en Git, casi todas ellas sólo añaden información a la base de datos de Git. Es muy complejo conseguir que el sistema haga algo que no se pueda deshacer, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía; pero después de confirmar una instantánea en Git, es muy complejo perderlo, especialmente si haces push, tu base de datos a otro repositorio con regularidad.

Los tres estados

Git tiene tres estados principales en los que se pueden encontrar los archivos: confirmado (committed), modificado (modified), y preparado (staged). Confirmado significa que los datos están almacenados de manera segura en tu base de datos local. Modificado significa que un archivo ha sido modificado pero todavía no se ha confirmado en la base de datos. Preparado significa que ha sido marcado un archivo modificado en su versión actual para que se añada en la próxima confirmación.
Esto nos lleva a las tres secciones principales de un proyecto de Git: el directorio de Git (Git directory), el directorio de trabajo (working directory), y el área de preparación (staging area).



Directorio de trabajo, área de preparación y directorio de Git.

El directorio de Git es donde Git almacena los metadatos y la base de datos de objetos para el proyecto. Es la parte más importante de Git, y es lo que se copia cuando se realiza una operación de clonado sobre un repositorio desde otro equipo.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para su posterior uso y/o modificación.

El área de preparación es un sencillo archivo, generalmente contenido en el directorio de Git, que almacena información acerca de lo que incluirá la próxima confirmación. Comúnmente llamado índice. Pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

El flujo de trabajo básico en Git es el siguiente:

1. Modificación de una serie de archivos en el directorio de trabajo.
2. Preparación de los archivos, añadido al área de preparación.
3. Confirmación de los cambios, lo que toma los archivos tal y como están en el área de preparación, y almacena esas instantáneas de manera permanente en el directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

REST

REST no es una tecnología, ni siquiera una arquitectura, es un estilo arquitectónico. Es un conjunto de restricciones a respetar cuando diseñamos la arquitectura de nuestros servicios web. Las restricciones propuestas por REST son las siguientes:

- REST no es RPC, sino orientado a recursos. Los servicios web no representan acciones, sino entidades de negocio. En vez de publicar verbos como por ejemplo “comprar”, se publican nombres como “carrito de la compra” o “pedido”. En este sentido podemos pensar en RPC como intentar definir la API de un sistema en base a procedimientos, es decir, es un paradigma procedural. Sin embargo REST define la API como un conjunto de recursos que representan objetos de negocio; este enfoque está mucho más cercano a la OO que a otra cosa.

- Cada recurso posee un identificador único universal (UUID o GUID) con el cual podemos hacer referencia a él. Estas referencias son accesibles a través de la API, a la que proporcionamos acceso a clientes. Para proceder al acceso al recurso puede utilizarse para crear una relación desde un recurso a otro. Crear estas relaciones es tan sencillo como incluir una referencia de un recurso a otro usando el UUID del último.

- La implementación, y la forma exacta en la que un recurso se representa internamente, debe ser privada y no accesible al exterior.

- Cada recurso tiene una interfaz, o conjunto de operaciones que admite. Basta saber el UUID del recurso para poder enviarle la operación que queremos realizar.

- La interfaz es homogénea para todos los recursos. Esto quiere decir que todos los recursos deben escoger las operaciones que soportan de entre un conjunto cerrado de acciones. Este conjunto de operaciones permitidas es una característica específica de cada arquitectura REST y no puede cambiarse. Como consecuencia no podemos inventar nuevas operaciones, sino que tenemos que modelar esas nuevas operaciones como recursos. El conjunto mínimo de operaciones que debe tener un sistema REST es “leer”, “actualizar”, “crear” y eliminar.

- Las operaciones se realizan mediante la transferencia del estado del recurso entre cliente y servidor. El cliente puede pedir que una copia del estado de un recurso sea transferido desde el servidor al cliente (leer), modificarlo, y mandar la copia modificada al servidor usando alguna de las operaciones de modificación permitidas (actualizar o crear).

- Las operaciones son stateless. Es decir, el resultado de una operación es independiente de la conversación que hayan mantenido el cliente y el servidor anteriormente. Como consecuencia de esto, toda petición debe llevar asociado el cliente que la realiza y el objeto al que hace referencia. De lo contrario, el servidor no podrá llevar a cabo las operaciones pertinentes.

Los recursos son multimedia, es decir, el estado de un recurso puede ser representado mediante distintos formatos. Por formato se entiende el formato concreto usado en la serialización del estado cuando se manda por red. Algunos ejemplos de formatos son XML, JSON, imagen JPEG, imagen GIF, etc. El estado de un recurso puede ser copiado desde un servidor al cliente o viceversa, pero por otro lado el formato usado para representar internamente el recurso es privado y desconocido por el cliente. Es de vital importancia, para que el sistema al completo funcione, que el cliente debe especificar al servidor que formatos entiende y viceversa, para ponerse de acuerdo en el conjunto de formatos a usar, según las necesidades ambas partes.

El acrónimo REST responde a “REpresentational State Transfer”. El estilo arquitectónico REST fue descrito por primera vez por Roy Thomas Fielding, el año 2000.

Los verbos HTTP

Una característica completamente alineada con REST del protocolo HTTP es el hecho de tener una interfaz uniforme para todos los recursos web. HTTP define un conjunto predefinido y cerrado de acciones o métodos HTTP. Es importante tener en cuenta que la propia especificación define los conceptos de seguridad e idempotencia, y clasifica los métodos conforme a estos dos criterios. Un método se considera seguro si no produce efectos secundarios. Por efecto secundario se entiende cualquier modificación del estado del servidor, o interacción de éste con cualquier otro sistema, que produzca efectos perceptibles por el usuario. Normalmente sólo los métodos que representan lectura se consideran seguros. Un método es idempotente si la ejecución repetida de éste, con exactamente los mismos parámetros, tiene el mismo efecto que si sólo se hubiera ejecutado una vez. Esta propiedad nos permite reintentar con seguridad una petición una y otra vez, y tener la seguridad de que la operación no se va a duplicar. Los métodos (o también llamados verbos) HTTP usados con más frecuencia son los siguientes:

VERBO	Seguro	Idempotente	Semántica
GET	Si	Si	Leer el estado del recurso
PUT	No	Si	Crear un recurso
POST	No	No	Cualquier acción genérica no idempotente
PATCH	No	Si	Actualizar un recurso
OPTIONS	Si	Si	Enumerar las opciones de un recurso
DELETE	No	Si	Eliminar un recurso

El verbo GET, se utiliza para leer el recurso, no lleva incluida ningún cuerpo, con parámetros adicionales; van incluidos en la uri del recurso. El método POST es objeto de frecuentes malentendidos. En los años 2000, cuando empezaron a usarse los formularios de HTML, los desarrolladores se dieron cuenta de que enviarlos por método GET no era tan seguro como hacerlo por post, ya que no permite cifrado de la uri. Es por ello, que se usaba el método para todo lo que conllevase acciones con formularios, indistintamente de su finalidad. En general se usa para crear un nuevo recurso, modificar uno existente o para ejecutar una acción genérica que no sea idempotente. Es cierto que la arquitectura REST es relativamente nueva, aunque está pensada desde los años 90. Este error, hoy día se ve replicado en todos los aspectos. Es importante seguir el estándar que marca el protocolo HTTP para que no haya ningún problema, ni entre servidores ni desarrolladores.

Comunicación TCP/UDP

Introducción

Cuando hablamos de aplicaciones en tiempo real(RTA), es necesario conocer la arquitectura de la red, para saber que debemos elegir para poder llevar a cabo la solución óptima al problema. Para ello, es necesario disponer de la información más precisa posible. En cuanto a especificación se refiere, la forma mejor forma para informarse, es a través de los RFCs.

Request for Comments: son una serie de publicaciones del grupo de trabajo de ingeniería de internet que describen diversos aspectos del funcionamiento de Internet y otras redes de computadoras, como protocolos, procedimientos, etc. y comentarios e ideas sobre estos. Cada RFC constituye un monográfico o memorando que ingenieros o expertos en la materia han hecho llegar al IETF, el consorcio de colaboración técnica más importante en Internet, para que éste sea valorado por el resto de la comunidad. De hecho, la traducción literal de RFC al español es "Petición de comentarios".
Estas publicaciones se remontan a 1969, cuando Steve Crocker inventó un sistema eficaz de hacer llegar las propuestas técnicas al resto de grupos de trabajo que experimentaban con ARPANET, la precursora de Internet.

Extracto del RFC 768(UDP)

Este Protocolo de Datagramas de Usuario (UDP: User Datagram Protocol) se define con la intención de hacer disponible un tipo de datagramas para la comunicación por intercambio de paquetes entre ordenadores en el entorno de un conjunto interconectado de redes de computadoras. Este protocolo asume que el Protocolo de Internet (IP: Internet Protocol) [1] se utiliza como protocolo subyacente.

Este protocolo aporta un procedimiento para que los programas de aplicación puedan enviar mensajes a otros programas con un mínimo de mecanismo de protocolo. El protocolo se orienta a transacciones, y tanto la entrega como la protección ante duplicados no se garantizan.
Las aplicaciones que requieran de una entrega fiable y ordenada de secuencias de datos deberían utilizar el Protocolo de Control de Transmisión (TCP: Transmission Control Protocol).

Extracto del RFC 793

El "protocolo de control de transmisión" ('Transmission Control Protocol', TCP) está pensado para ser utilizado como un protocolo 'host' a 'host' muy fiable entre miembros de redes de comunicación de computadoras por intercambio de paquetes y en un sistema interconectado de tales redes.

Este documento describe las funciones que debe realizar el protocolo de control de transmisión, el programa que lo implementa, y su interfaz con los programas o usuarios que requieran de sus servicios.

1.1. Motivación

Los sistemas de comunicación entre computadoras están jugando un papel cada vez más importante en entornos militares, gubernamentales y civiles. Este documento centra principalmente su atención en los requisitos militares de comunicación entre computadoras, especialmente la robustez bajo comunicaciones no plenamente fiables y la disponibilidad ante congestiones, aunque muchos de estos problemas pueden encontrarse igualmente en los sectores civil y gubernamental.

A la par que las redes estratégicas y tácticas de comunicación entre computadoras están siendo desarrolladas y desplegadas, es esencial proporcionar medios de interconexión entre ellas y proporcionar protocolos estándares de comunicación entre procesos que puedan soportar un amplio rango de aplicaciones. Anticipando la necesidad de tales estándares, la subsecretaría de defensa del congreso de los diputados para la investigación e ingeniería ('the Deputy Undersecretary of Defense for Research and Engineering') ha declarado el protocolo de transmisión de control (TCP) descrito aquí como la base para la estandarización de los protocolos de comunicación entre procesos, dentro del ámbito de todo el Departamento de Defensa (DoD).

TCP es un protocolo orientado a la conexión, fiable y entre dos extremos, diseñado para encajar en una jerarquía en capas de protocolos que soportan aplicaciones sobre múltiples redes. TCP proporciona mecanismos para la comunicación fiable entre pares de procesos en computadoras 'host' ancladas en redes de comunicación de computadoras distintas, pero interconectadas. Se hacen muy pocas suposiciones sobre la fiabilidad de los protocolos de comunicación por debajo de la capa de TCP. TCP sólo supone que puede acceder a un servicio de transmisión de datagramas simple, aunque en principio poco fiable, de los protocolos del nivel inferior. En principio, TCP debería ser capaz de operar encima de un amplio espectro de sistemas de comunicaciones que incluye desde conexiones por cables fijos ('hard-wired conections') hasta redes de intercambio de paquetes o redes de circuitos conmutados.

TCP se basa en los conceptos descritos primeramente por Cerf y Kahn en [1]. TCP encaja en una arquitectura de protocolos en capas justo por encima del protocolo de internet [2], protocolo básico que proporciona un medio para TCP de enviar y recibir segmentos de longitud variable de información envuelta en "sobres" de datagramas de internet. El datagrama de internet proporciona un medio de direccionar TCPs de origen y de destino situados en redes diferentes. El protocolo de internet también trata con la fragmentación y el reensamble de segmentos de TCP que sean necesarios para conseguir el transporte y la entrega sobre múltiples redes y las puertas de enlace que las interconectan. El protocolo de internet también lleva información sobre la prioridad, clasificación de seguridad y compartimentación de los segmentos de TCP, de tal forma que esta información pueda ser comunicada de extremo a extremo entre múltiples redes.

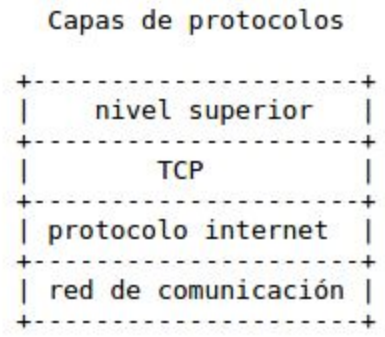


Figura 1

Gran parte de este documento se ha escrito dentro del contexto de la implementaciones de TCP que son corresidentes con protocolos de más alto nivel en la computadora anfitriona. Algunos sistemas de computadoras se conectarán a las redes vía computadoras intermediarias ('front-end computers') que alojan las capas de protocolos TCP e internet, a la vez que el software específico de redes. Esta especificación de TCP describe una interfaz con los protocolos de mayor nivel que resulta ser implementable incluso para el caso de computadoras intermediarias, siempre y cuando se implemente un adecuado protocolo entre el 'host' y la computadora intermediaria.

1.2. Ámbito

TCP está pensado para proporcionar un servicio fiable de comunicación entre procesos en un entorno con múltiples redes. TCP está pensado para ser un protocolo de 'host' a 'host' de uso común en redes múltiples.

1.3. Sobre este documento

Este documento representa una especificación del comportamiento requerido a cualquier implementación de TCP, tanto en sus interacciones con protocolos de más alto nivel como con sus interacciones con otros TCP. El resto de esta sección ofrece una muy breve visión de las interfaces y operaciones del protocolo. La sección 2 resume los presupuestos de corte filosófica para el diseño de TCP. La sección 3 ofrece tanto una descripción detallada de las acciones que se le exigen a TCP cuando varios eventos acontecen (llegada de nuevos segmentos, llamadas de usuario, errores, etc.) así como los detalles de los formatos de los segmentos TCP.

1.4. Interfaces

TCP presenta interfaz por un lado con el usuario o los procesos de aplicación y por el otro con un protocolo de más bajo nivel como es el protocolo de internet (IP).

La interfaz entre un proceso de aplicación y TCP se ilustrará con un detalle razonable. Esta interfaz consiste en un conjunto de llamadas, de forma muy similar a las llamadas que un sistema operativo proporciona a los procesos de aplicación para manipular ficheros. Por ejemplo, hay llamadas para abrir y cerrar conexiones y para enviar y recibir datos por las conexiones establecidas. Se exige también que TCP pueda comunicarse asíncronamente con los programas de aplicación.
Aunque se deja considerable libertad a los fabricantes de implementaciones de TCP a la hora de diseñar las interfaces que sean apropiadas para el entorno de un sistema operativo particular, se exige un mínimo de funcionalidad en la interfaz TCP/usuario de cualquier implementación válida.

La interfaz entre TCP y el protocolo de nivel inferior queda esencialmente sin especificar, exceptuando el hecho de que se asume que hay un mecanismo por el cual los dos niveles pueden pasar información asíncronamente el uno al otro. Típicamente, se espera que el protocolo de nivel inferior especifique esta interfaz. Se ha diseñado TCP para trabajar en un entorno muy genérico de redes interconectadas. El nivel inferior que se asumirá a lo largo de este documento es el protocolo de internet [2].

1.5. Operación

Como se ha hecho notar más arriba, el propósito principal de TCP consiste en proporcionar un servicio de conexión o circuito lógico fiable y seguro entre pares de procesos. Para proporcionar este servicio encima de un entorno de internet menos fiable, el sistema de comunicación requiere de mecanismos relacionados con las siguientes áreas:

Transferencia básica de datos
Fiabilidad
Control de flujo
Multiplexamiento
Conexiones
Prioridad y seguridad

La operación básica de TCP en cada uno de estas áreas se describe en los siguiente párrafos:

Transferencia básica de datos:

TCP es capaz de transferir un flujo continuo de octetos en cada sentido entre sus usuarios empaquetando un cierto número de octetos en segmentos para su transmisión a través del sistema de internet. En general, los módulos de TCP deciden cuándo bloquear y enviar datos según su propia conveniencia.

Algunas veces los usuarios necesitan estar seguros de que todos los datos que habían entregado al módulo de TCP han sido transmitidos. Para este propósito se define una función 'push' ("enviar inmediatamente"). Para asegurar que los datos entregados al módulo de TCP son realmente transmitidos, el usuario emisor debe indicarlo mediante la función 'push'. Un 'push' en un cierto instante causa que los módulos de TCP envíen y entreguen inmediatamente al usuario receptor los datos almacenados hasta ese instante. El instante exacto en que se ejecuta la función 'push' podría no ser visible para el usuario receptor. Tampoco la función 'push' proporciona una marca de límite de registros.

Fiabilidad:

El módulo de TCP debe poder recuperar los datos que se corrompan, pierdan, dupliquen o se entreguen desordenados por el sistema de comunicación del entorno de internet. Esto se consigue asignando un número de secuencia a cada octeto transmitido, y exigiendo un acuse de recibo (ACK, N.T.:del inglés 'acknowledgment') del módulo de TCP receptor. Si no se recibe un ACK dentro de un cierto plazo de expiración prefijado, los datos se retransmiten. En el receptor, se utilizan los números de secuencia para ordenar correctamente los segmentos que puedan haber llegado desordenados y para eliminar los duplicados. La corrupción de datos se trata añadiendo un campo de suma de control ('checksum') a cada segmento transmitido, comprobandose en el receptor y descartando los segmentos dañados.

En tanto en cuanto los módulos de TCP continúen funcionando adecuadamente y el sistema de internet no llegue a quedar particionado de forma completa, los errores de transmisión no afectarán la correcta entrega de datos. TCP se recupera de los errores del sistema de comunicación de internet.

Flujo de control:

TCP proporciona al receptor un medio para controlar la cantidad de datos enviados por el emisor. Esto se consigue devolviendo una "ventana" con cada ACK, indicando el rango de números de secuencia aceptables más allá del último segmento recibido con éxito. La ventana indica el número de octetos que se permite que el emisor transmita antes de que reciba el siguiente permiso.

Multiplexamiento:

Para permitir que muchos procesos dentro de un único 'host' utilicen simultáneamente las posibilidades de comunicación de TCP, el módulo de TCP proporciona una serie de direcciones o puertos dentro de cada 'host'. Concatenadas con las direcciones de red y de 'host' de la capa de comunicación internet conforman lo que se denomina una dirección de conector ('socket'). Un par de direcciones de conector identifica de forma única la conexión. Es decir, un conector puede utilizarse simultáneamente en múltiples conexiones.
La asignación de puertos a los procesos se gestiona de forma independiente en cada 'host'. Sin embargo, resulta de la máxima utilidad asignar a los procesos más utilizados frecuentemente (i.e., un gestor de registros ('logger') o un servicio compartido) conectores fijos que se hacen conocer de forma pública. Estos servicios pueden entonces ser accedidos a través de direcciones conocidas públicamente. El establecimiento y aprendizaje de las direcciones de los puertos de otros procesos puede involucrar otros mecanismos más dinámicos.

Conexiones

La fiabilidad y los mecanismos de control de flujo descritos más arriba exigen que los módulos de TCP inicialicen y mantengan una información de estado para cada flujo de datos. La combinación de esta información, incluyendo las direcciones de los conectores, los números de secuencia y los tamaños de las ventanas, se denomina una conexión. Cada conexión queda especificada de forma única por un par de conectores que corresponden con sus dos extremos.

Cuando dos procesos desean comunicarse, sus módulos de TCP deben establecer primero una conexión (inicializar la información de estado en cada lado). Cuando la comunicación se ha completado, la conexión se termina o cierra con la intención de liberar recursos para otros usos.

Como las conexiones tienen que establecerse entre 'hosts' no fiables y sobre un sistema de comunicación internet no fiable, se utiliza un mecanismo de acuerdo que usa números de secuencia basados en tiempos de reloj para evitar una inicialización errónea de las conexiones.

SaaS/PaaS/IaaS/(DaaS)

En este tipo de computación todo lo que puede ofrecer un sistema informático se ofrece como servicio, de modo que los usuarios puedan acceder a los servicios disponibles "en la nube de Internet" sin conocimientos (o, al menos sin ser expertos) en la gestión de los recursos que usan. Según el IEEE Computer Society, es un paradigma en el que la información se almacena de manera permanente en servidores de Internet y se envía a cachés temporales de cliente, lo que incluye equipos de escritorio, centros de ocio, portátiles, etc.
La computación en la nube son servidores desde Internet encargados de atender las peticiones en cualquier momento. Se puede tener acceso a su información o servicio, mediante una conexión a internet desde cualquier dispositivo móvil o fijo ubicado en cualquier lugar. Sirven a sus usuarios desde varios proveedores de alojamiento repartidos frecuentemente por todo el mundo. Esta medida reduce los costos, garantiza un mejor tiempo de actividad y que los sitios web sean invulnerables a los delincuentes informáticos, a los gobiernos locales y a sus redadas policiales.

"Cloud computing" es un nuevo modelo de prestación de servicios de negocio y tecnología, que permite incluso al usuario acceder a un catálogo de servicios estandarizados y responder con ellos a las necesidades de su negocio, de forma flexible y adaptativa, en caso de demandas no previsibles o de picos de trabajo, pagando únicamente por el consumo efectuado, o incluso gratuitamente en caso de proveedores que se financian mediante publicidad o de organizaciones sin ánimo de lucro.

El cambio que ofrece la computación desde la nube es la posibilidad de aumentar el número de servicios basados en la red. Esto genera beneficios tanto para los proveedores, que pueden ofrecer, de forma más rápida y eficiente, un mayor número de servicios, como para los usuarios que tienen la posibilidad de acceder a ellos, disfrutando de la ‘transparencia’ e inmediatez del sistema y de un modelo de pago por consumo. Así mismo, el consumidor ahorra los costes salariales o los costes en inversión económica (locales, material especializado, etc.).
La Computación en la nube consigue aportar estas ventajas, apoyándose sobre una infraestructura tecnológica dinámica que se caracteriza, entre otros factores, por un alto grado de automatización, una rápida movilización de los recursos, una elevada capacidad de adaptación para atender a una demanda variable, así como virtualización avanzada y un precio flexible en función del consumo realizado, evitando además el uso fraudulento del software y la piratería.

La computación en nube es un concepto que incorpora el software como servicio, como en la Web 2.0 y otros conceptos recientes, también conocidos como tendencias tecnológicas, que tienen en común el que confían en Internet para satisfacer las necesidades de cómputo de los usuarios.

Infraestructure as a Service (IaaS):

- Profiere a la solución software de un control total sobre la máquina, ya que dispone de capacidad de proceso (CPU) y almacenamiento.

- Gracias a la automatización, existe la posibilidad de escalar los recursos requeridos, tanto aumentar como disminuir.

- Los servicios de este tipo suelen realizar las tareas a través de sistemas operativos virtualizados. Razón por la cual, resulta tan sencillo aumentar los recursos del sistema.

- Utilizar este tipo de servicios, no permite probar la ejecución de una solución software con altas prestaciones a unos precios muy bajos, ya que solo se paga por lo que se usa. Frente a la solución tradicional, comprar los equipos, es ideal para llevar a cabo pruebas a un bajo coste.

- Ejemplo: la conocida EC2 de Amazon y Azure de Microsoft

Este tipo de soluciones nos permiten ejecutar determinados servicios con las prestaciones que necesitemos a un bajo coste. Ideal para tareas con un alto consumo en CPU, donde es el programador, el encargado de hacer rendir la solución a un menor nivel. Para acceder, el proveedor, normalmente, pone a disposición del usuario un terminal de ssh con plenos derechos de modificación.

Platform as a Service (PaaS):

- es el siguiente paso, se proporciona además un servidor de aplicaciones (donde se ejecutarán nuestras aplicaciones) y una base de datos.

- Podremos instalar las aplicaciones y ejecutarlas. Normalmente hay que seguir una serie de restricciones para poder desarrollar estas para un proveedor (por ejemplo en cuanto a los lenguajes de programación).

- Estos tipos de servicio incluyen paquetes de software que dan acceso a muchos tipos de instalaciones. Por ejemplo, módulos de Redis, de sistemas de caché, etc.

- Ejemplo: Heroku, Openshift.

Este tipo de solución nos permite ejecutar aplicaciones del lado del servidor que no precisan de un control tan exhaustivo del rendimiento. Los servicios que disponen de esta modalidad, suelen utilizarse para desplegar servidores de aplicaciones de variedad de modalidades. Por ejemplo, un servidor de tipo REST que sería capaz de dar servicio a una App. Para acceder, el proveedor, normalmente, pone a disposición del usuario un terminal de ssh con comandos personalizados, que desencadenan acciones automatizadas, como por ejemplo, arrancar la instancia, parar, consultar estadísticas, etc.

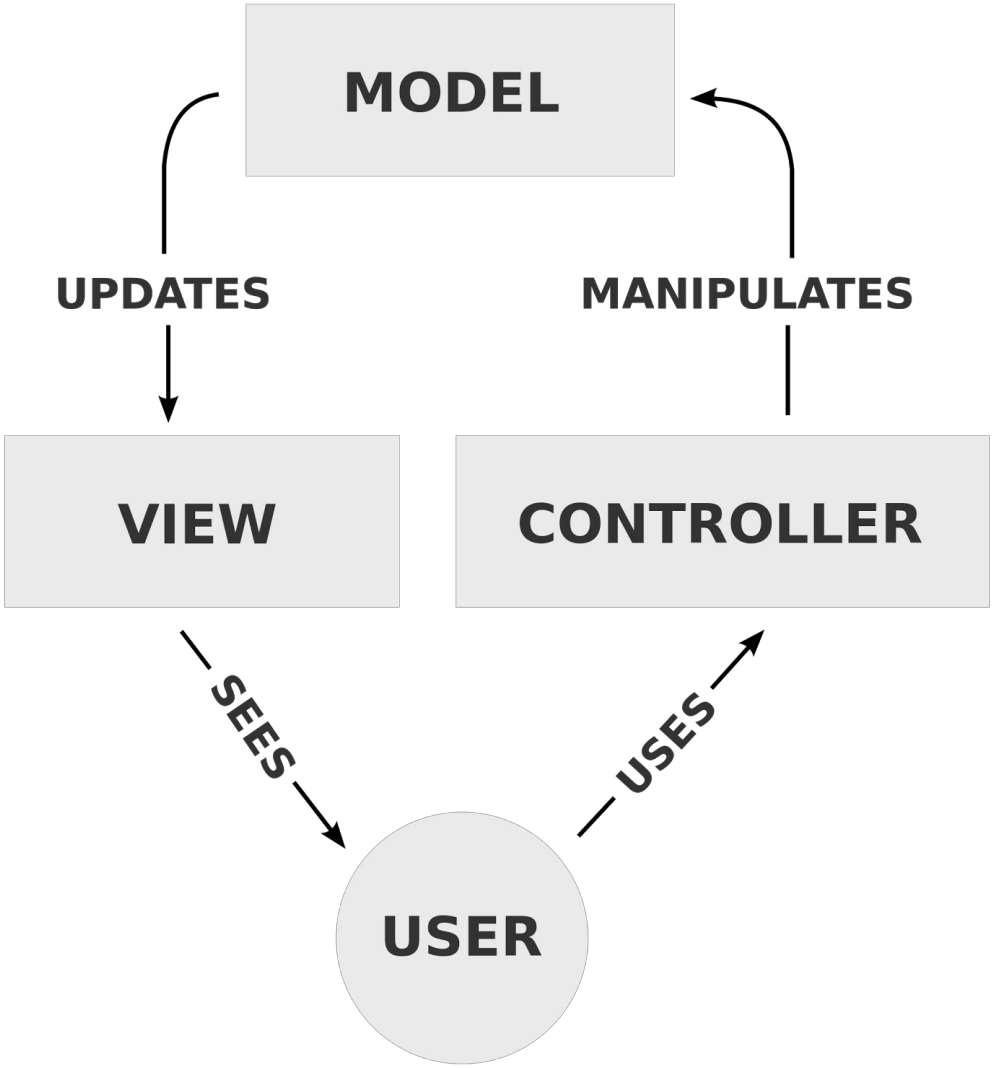
Software as a Service (SaaS):

Es lo que comúnmente se identifica con “cloud”. Es una aplicación para el usuario final donde paga un alquiler por el uso de software. No es necesario adquirir un software en propiedad (como Microsoft Office), instalarlo, configurarlo y mantenerlo.

Actualmente,siguen surgiendo ramificaciones de PaaS especializadas, tales como DaaS. Donde la plataforma, de lo único que se encarga es de alojar la base de datos, y ofrece al desarrollador la posibilidad de acceder mediante dos opciones:

- A través del protocolo de la base de datos. Por ejemplo: Mysql, Postgres, Oracle, etc.
- Algunos proveedores, han creado una API más rica, que permite además extraer más información, de forma ordenada. Todo ello a través de REST.
- Ejemplo: Google Docs, Office365 o Dropbox.

MVC



Modelos
Es la capa donde se trabaja con los datos, por tanto contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos estarán ubicados habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a ellos y harán las correspondientes operaciones para obtener la información deseada.
No obstante, cabe mencionar que cuando se trabaja con MCV lo habitual también es utilizar otras librerías como PDO o algún ORM como Doctrine, que nos permiten trabajar con abstracción de bases de datos y persistencia en objetos. Por ello, en vez de usar directamente sentencias SQL, que suelen depender del motor de base de datos con el que se esté trabajando, se utiliza un dialecto de acceso a datos basado en clases y objetos.

Vistas
Las vistas, como su nombre nos hace entender, contienen el código de nuestra aplicación que va a producir la visualización de las interfaces de usuario, o sea, el código que nos permitirá renderizar los estados de nuestra aplicación. Las vistas son las encargadas de transformar en última instancia los modelos para presentarlos al usuario de la manera deseada.
En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generará la salida, tal como nuestra aplicación requiera.

Controladores
Contiene el código necesario para responder a las acciones que se solicitan en la aplicación. Es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación. Sin embargo, su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas para implementar las diversas necesidades del desarrollo.

Arquitectura de aplicaciones MVC
Los controladores, con su lógica de negocio, hacen de puente entre los modelos y las vistas. Pero además en algunos casos los modelos pueden enviar datos a las vistas. A continuación se expone el flujo de trabajo característico en un esquema MVC, relacionado con el mundo web.

1. El usuario realiza una solicitud a un sitio web. Esa solicitud le llega al controlador.
2. El controlador comunica tanto con modelos como con vistas. A los modelos les solicita datos transformaciones de los mismos. A las vistas les solicita la salida correspondiente, una vez se hayan realizado las operaciones pertinentes según la lógica del negocio.
3. Para producir la salida, en ocasiones las vistas pueden solicitar más información a los modelos. En ocasiones, el controlador será el responsable de solicitar todos los datos a los modelos y de enviarlos a las vistas, haciendo de puente entre ambos. Sería corriente tanto una cosa como la otra, todo depende de la implementación.
4. Las vistas generan lo que el usuario verá. Aunque el que sirva las vistas, siempre será el controlador.

Lógica de negocio / Lógica de la aplicación
Es un conjunto de reglas que se siguen en el software para reaccionar ante distintas situaciones. En una aplicación el usuario se comunica con el sistema por medio de una interfaz, pero cuando acciona esa interfaz para realizar acciones con el sistema, se ejecutan una serie de procesos que se conocen como la lógica del negocio. Este es un concepto de desarrollo de software que puede ser representado en cualquier lenguaje.

La lógica del negocio, independientemente de marcar un comportamiento cuando ocurren cosas dentro de un software, también tiene normas sobre lo que se puede hacer y lo que no. Eso también se conoce como reglas del negocio. En el MVC la lógica del negocio queda del lado de los modelos. Ellos son los que deben saber cómo operar en diversas situaciones y las acciones que pueden permitir que ocurran en el proceso de ejecución de una aplicación.

Sin embargo existe otro concepto que se usa en la terminología del MVC que es la "lógica de aplicación", que es algo que pertenece a los controladores. Esta lógica es la que representa lo que ha de hacer el sistema para solicitar los datos adecuados al modelo y qué hacer con ellos. Este concepto, Lógica de aplicación, no está tan extendido entre todos los teóricos, pero puede ayudar a comprender dónde está cada responsabilidad del sistema, es decir, demarca los límites entre lo que se puede hacer o no, en cada capa. Cada una tiene su cometido, para que todo funcione correctamente, y el código sea mantenible, escalable y desacoplado, tiene como requisito indispensable, seguir las reglas del MVC.

Desarrollo

Jerarquía

El framework está dividido en tres partes, Servidor, nodo y bot. Cada uno de ellos se encargan de tareas muy diferentes. La división entre ellos puede aplicarse a distintos grados. A través de la ejecución en diferentes proveedores, diferentes cuentas, o incluso en la misma cuenta con instancias diferentes.

- Servidor: encargado de mostrar al usuario una interfaz donde el usuario pueda administrar las tareas. A través de esa interacción se actuará en consecuencia para derivar el trabajo a los nodos correspondientes.
- Nodo: no se tiene acceso al mismo directamente. Es el puente de interconexión entre la fuerza de trabajo efectiva(bot) y el servidor. Se encarga además de coordinar las soluciones independientes de cada uno de los bots.
- Bot: es el último eslabón de la cadena que hace el trabajo que requiere de la capacidad de cómputo.

La tarea a desempeñar ha de ser divisible en unidades independientes de trabajo. Pues de lo contrario, el trabajo no se podría derivar a los bots encargados de resolver el problema. Este aspecto ha de ser solventado por el desarrollador, pues el framework no está diseñado para tomar este tipo de decisiones.

Modo de Funcionamiento

El usuario accede al panel de administración para configurar una nueva tarea. Para ello proporciona toda la información necesaria en el servidor, a través de su panel web. Acto seguido, el servidor envía el trabajo a los nodos, y éstos, se encargan de dividir el trabajo proporcionalmente entre los bots disponibles, bajo las condiciones impuestas por el usuario. Este es un aspecto crítico, ya que una mala decisión puede acarrear retardos en el proceso de ejecución del framework, sino algo peor, la ejecución errónea de la solución.

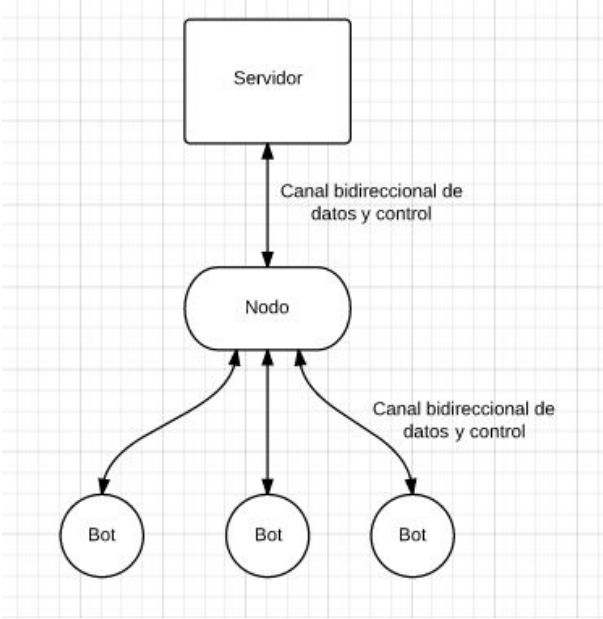
Posteriormente, todos reciben su trabajo, con una fecha determinada. De esta forma, todos trabajarán a la vez. Sería posible usar fechas acumulativas entre los diferentes bots, para hacer facilitar el proceso de unión de soluciones al nodo superior. Este tipo de restricciones, también deberán ser elegidas por el desarrollador. A mayor número de bots al cargo de cada nodo, mayor será la carga de trabajo que habrá de desempeñar. Los problemas provenientes de estos asuntos serán estudiados con mayor profundidad a continuación.

Por último, realizar el trabajo para el que han sido programados los bots, comunicarán la solución a la capa superior, en este caso, cada uno a su nodo correspondiente. Los nodos serán los encargados de unir las soluciones proporcionadas por los bots. Acto seguido, se comunicarán entre nodos para unir las soluciones. Este es un paso crítico, ya que puede producir muchos problemas. Se ha de establecer una condición de solución clara, para evitar confusión entre los nodos. En el caso de despliegues con grandes cantidades de nodos y bots, sería necesaria separar la capa de nodos, en primarios y secundarios. Los primarios, encargados de unir la solución final de cada nodo a su cargo secundario. Y éstos últimos, unir las soluciones proporcionados por los bots a su cargo.

Descripción

El sistema, solo podrá llevar a cabo una tarea en su totalidad. Pues, la consecución de tareas diferentes en mismos estados de tiempo, dificultaría, sino imposibilitaría la comunicación y el buen funcionamiento del sistema en su totalidad, haciéndolo inestable. La capacidad de cómputo de cada nodo, puede ser calculada de forma aproximada, con las estadísticas de cómputo particulares, para establecer un tiempo aproximado de ejecución. El número de bots aproximados que un nodo puede llegar a manipular, viene marcado por las variables: capacidad de cómputo de los nodos, retrasos en la comunicación, tiempo de ejecución de las órdenes y unión de las comunicaciones.

La jerarquía de la aplicación será la siguiente, dividida en tres capas, Servidor, nodo y bot.



Como se puede observar en la imagen anterior, las tres capas están bien distinguidas. Cada capa tiene su tarea bien definida. Donde la comunicación es vital entre elementos. Además, se sigue un patrón ascendente, donde el extremo inferior, no puede comunicar con el extremo superior y viceversa. La separación entre capas proporciona desacoplamiento de las capas, y la posibilidad de escalabilidad.

Servidor(Vista)

El servidor es un elemento intermedio entre el usuario y el el sistema en su conjunto. Es el encargado de mostrar la solución al usuario, pero sin intervenir en ella; como si de la vista del MVC se tratase. Por otra parte, también permite al usuario, a través de su interfaz comunicar al resto del sistema las tarea que se han de llevar a cabo, y bajo qué condiciones.

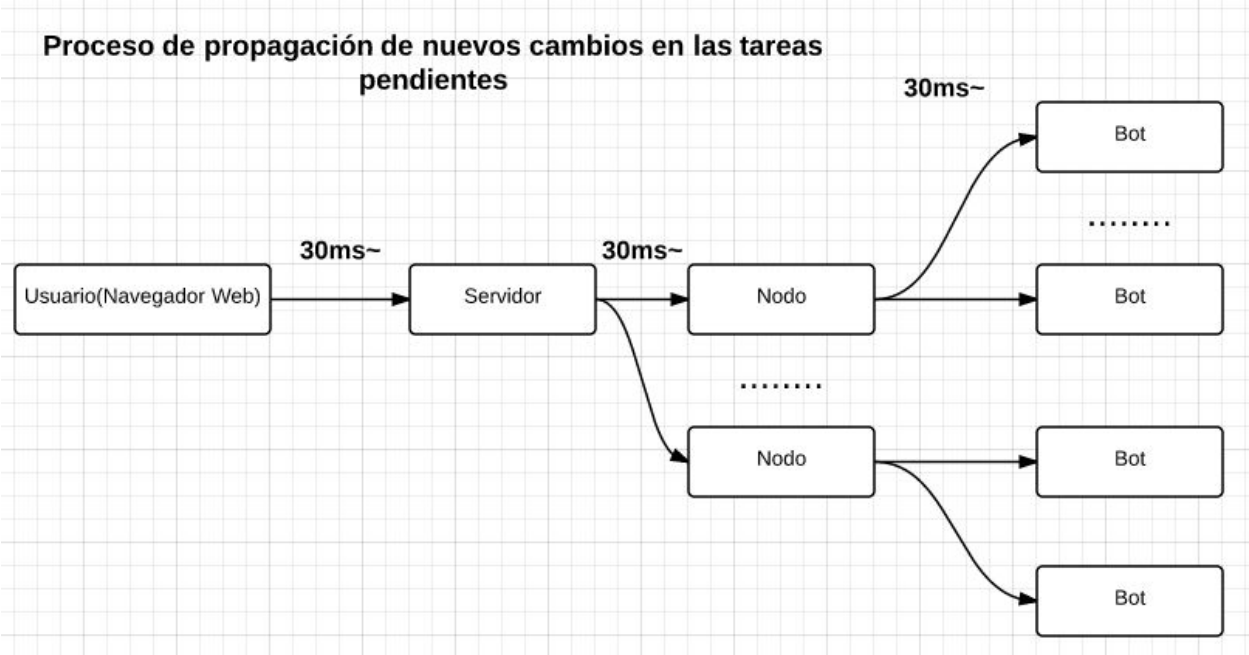
El servidor tiene dos tareas bien definidas:

- Administración: esta parte, es la encargada de interactuar con el usuario a través de una interfaz web. Es posible segregar en MVC la parte del servidor web, pudiendo separar los controladores, encargados de interactuar entre vista y controlador. Vistas, que permitirán mostrar al usuario el estado del sistema, y modelos, que almacenarán las tareas, así como su estado.

- Comunicación: las acciones que el usuario lleve a cabo en la interfaz, tienen una repercusión en el sistema. El servidor es el encargado de propagar esas acciones a las capas inferiores. Cuando el sistema es menor a un tamaño dado, la comunicación puede realizarse a través de las capas superiores del modelo TCP/IP, perdiendo velocidad, y ganando a cambio las ventajas que ofrecen las capas inferiores respecto de la capa de aplicación, tales como control de flujo, control de errores, fragmentación de paquetes.

Limitaciones

El servidor, en este caso, tendrá una carga moderada, la que supone servir e interactuar con un cliente web. Tratándose de un único cliente, en este caso, sólo habría un administrador o un grupo reducido de personas que podrían acceder al servidor web. Esto hace que la carga sea la misma para un grupo reducido de personas que accederán a él. La ejecución de las acciones que un cliente realiza, se han de estudiar por separado, en función del número de recursos que involucre y el número elementos que se vean afectados por el cambio. Es decir, si se actualiza la fecha en la que debe ejecutar la solución software todos los bots, tenemos muchos factores que pueden afectar al tiempo total. Si partimos desde el punto en el cual, el cliente ya tiene la web renderizada en su navegador, y se dispone a enviar un formulario con los datos ya introducidos:



Suponemos los siguientes factores:

- El tiempo que el cliente tarda en enviar la respuesta al servidor, procesa la petición, y responde al cliente. En este último paso, la respuesta al cliente no es relevante para calcular el tiempo total.

- El servidor envía las peticiones pertinentes a los nodos que tiene a su disposición de manera asíncrona. El tiempo de cada conexión, tiene un retardo entre petición y petición de microsegundos (el tiempo que tarda en ejecutar la siguiente instrucción de programa, sin contar los procesos internos del intérprete y/o sistema operativo), que obviaremos por razones técnicas, ya que el número de nodos usados para realizar las pruebas no es elevado. Para un tamaña elevado de nodos, si sería necesario calcular todos los costes que surgen desde la n-1 hasta la n petición. Pudiendo retardar en gran medida la última petición entre servidor y nodo.

- Al igual que en el paso anterior, tomaremos como tiempo de conexión, 30ms. En este caso, no es necesario tener un gran número de nodos para que el tiempo entre llamada y llamada afecta al cómputo total de tiempo. Estos tiempos deben ser calculados en cada sistema en particular, ya que son muy sensibles a las características del hardware, así como la solución que ha implementado el proveedor de Servicios.

Con un coste total de:

30ms(petición HTTP) + 30ms(servidor) + 30ms(nodo) + tiempo_opreacion * nodos_totales(micros)

A esto se le ha de sumar la carga que supone comunicar las acciones a las capas inferiores.

<u>Requerimientos</u>

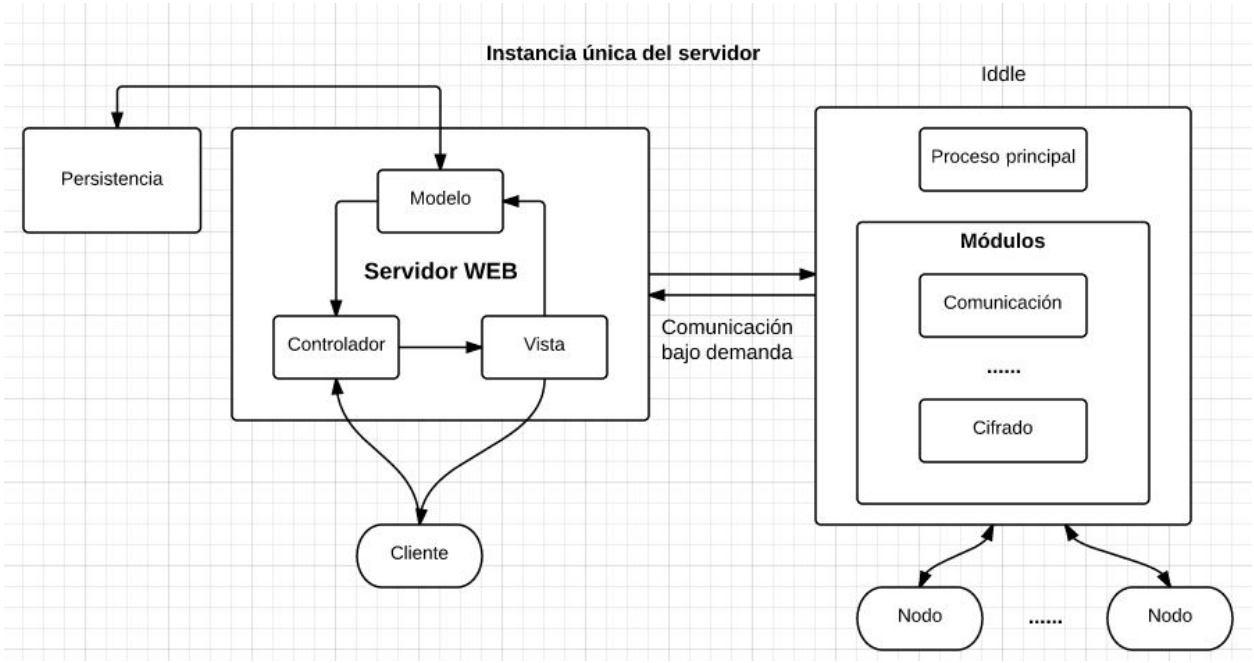
- Disponer del intérprete de node, para poder ejecutar la instancia del programa.

- Abrir/cerrar puertos.
    - Poner en escucha el servidor web, para poder servir el panel de administración.
    - (Alternativo)API REST para poder comunicar con el servidor a través de diferentes plataformas.
    - Comunicar con los nodos.

- (Alternativo)Instalación de módulos adicionales.

Especificaciones

Composición de la instancia Servidor:



Como se observa en la imagen anterior, el servidor está compuesto por dos instancias. Por una parte, el servidor web encargado de la administración, y por otra parte, la lógica que permite comunicar a las capas inferiores las acciones que se pretende realizar.

El modelo está directamente conectado a una instancia llamada Persistencia, que corresponde al contenedor de toda la información. En este caso, el framework viene predefinido con una base de datos no relacional. Este tipo de bases de datos son perfectas para el problema presentado, ya que permiten mantener una estructura variable y dinámica. Bajo condiciones cambiantes, podemos cambiar la estructura de los datos almacenados, para darles usos diferentes. El framework viene con un servidor web y una base de datos no relacional, preparada para funcionar sin ningún tipo de problema. El programador siempre es capaz de realizar los cambios que crea oportunos.

Para aumentar el rendimiento, la segunda parte, la de la comunicación, permanecerá en estado de espera, hasta que tenga acciones que realizar, ya sean, en sentido ascendente(de servidor a nodo), como descendente(de nodo a servidor). Si el servidor se ejecuta en una única instancia, para redes con pocos nodos y, en consecuencia, pocos bots, la comunicación entre las capas resulta más fluida y rápida. Si se segregan estos dos aspectos del servidor, sería necesaria llevar a cabo una sincronización entre la comunicación y el servidor web. Esto supondría lógica adicional, y para redes de un tamaño pequeño, no es necesaria tal infraestructura.

Servidor web(Administración)

Este aspecto es a libre elección del desarrollador, como base para este punto, podemos destacar unos factores clave que pueden ayudar a alcanzar una solución eficaz:

- Interfaz de usuario rica y simple
- Uso de plantillas de terceros para conseguir un entorno limpio
- Test unitarios del lado del cliente
- Uso de librerías y frameworks ampliamente utilizadas, en cualquier lenguaje, para evitar la duplicidad de código(bootstrap, AngularJS, FontAwesome, etc)

Como sugerencia de agrupación de utilidades, se nombran los menús, con las sus diferentes funcionalidades:

- Registros (Encargado de visualizar el comportamiento del sistema en su conjunto)
- Nodos (Operaciones con los nodos, crear, modificar, desactivar, etc)
- Bots (Operaciones con los bots, crear, modificar, desactivar, etc)
- Tareas (Operaciones con las tareas, crear, modificar, borrar, consultar)
- Estadísticas (Monitorizar todas las soluciones con datos en tiempo real)
- Configuración (Desde aquí podremos modificar configuraciones del sistema)

Comunicación

El servidor de comunicación se particulariza por su gran modularidad. Todas las acciones que es capaz de realizar, parten desde un bucle principal. El módulo principal realiza acciones que residen en los módulos activados en él. Este aspecto confiere la probabilidad de cambiar la funcionalidad de toda la infraestructura sin afectar en nada a todo lo demás. El programador, es capaz, por ejemplo, de modificar el tipo de comunicación únicamente, cambiando el módulo encargado de hacer conexión y responder.

- Cómo se comunica

La comunicación entre el servidor y los nodos, se lleva a cabo mediante peticiones HTTP, a través de una API REST. Su simplicidad y potencia, ligadas del estándar que la preceden conceden una robustez y estabilidad al conjunto del aplicativo. Esto es así debido a las ventajas que aporta la comunicación sobre la capa de aplicación, con las ventajas que adquiere a través el protocolo TCP, en este caso, de control de flujo, control de errores, fragmentación de paquetes, etc. En el caso de precisarse una comunicación continua entre capas, se puede desarrollar un módulo para comunicación en RTA, sustituyendo el módulo de comunicación.

- El proceso principal

El hilo principal del servidor de comunicación, se encarga de realizar la sincronización de todos los cambios realizados entre el usuario, en el panel web, y los nodos, cuando han realizado su trabajo. Frente a soluciones tradicionales del software, con el lenguaje javascript, podemos combinar un bucle de iteración, donde el software es capaz de decidir las acciones a realizar según las tareas a desempeñar, con las acciones callback, así como promises. La esencia del lenguaje, es añadir manejadores a los distintos eventos que ocurran en el sistema, para poder gestionarlos. Es decir, instanciar un recurso, apuntarlo en la lista de manejadores de ese evento en el intérprete de node, y esperar que ocurra. En el momento en el cual ese evento, es detectado, node ejecutará el manejador que tiene asociado. De esta forma, no es necesario tener un bucle infinito esperando que ocurra, ya que node, lo ejecutará cuando ocurra. Esta metodología de trabajo, permite realizar acciones cuando no se está usando la cpu, incrementando con ello, su uso y por consiguiente, el rendimiento.

- Módulos

Comunicación:

este módulo es el encargado de dos acciones, realmente importantes: enviar y recibir.

Cifrado:

este módulo es llamado previamente al envío de información, y posteriormente de la recepción de respuestas, para descifrar el contenido.

Pipe:

Pipening entre BACKEND Y SERVIDOR WEB.

- Acciones HTTP

Cada recurso, tendrá unas acciones asociadas, que pueden ser invocadas a través de la URI única para cada uno de ellos. A continuación se detallan las acciones que podrán ser llevadas a cabo por el proceso de comunicación:

Sobre la seguridad: para que el proceso de comunicación disponga de **confidencialidad**, **integridad**, **disponibilidad y no repudio**, es conveniente que todo tipo de comunicación entre capas siempre sea a través del tls/ssl. En este caso, https. Para evitar la complejidad añadida a este aspecto, no se han tomado este tipo de medidas. En un entorno productivo, este tipo de medidas son completamente obligatorias. Como se vio anteriormente, la modificación del flujo del aplicativo, no se ve afectada por el cambio de uno o más módulos. Gracias a su modularidad, existe la posibilidad de modificar el comportamiento de la comunicación, en este caso, el módulo *comunicación*.

Cosas a tener en cuenta: cuando se expone la uri que va ligada a un recurso, habitualmente, para distinguir el nombre de los parámetros de de la propia URI, se expresa con la notación *:atributo.* Para no extender la explicativa más de lo necesario, el significado de todos y cada uno de los campos devueltos por la respuesta, serán citados en los anexos. Así como la estructura de directorios, conjuntamente con los archivos de configuración.

- Tareas

La tarea es el trabajo que ha de realizar cada bot. Se compone de un archivo escrito en lenguaje javascript, que hereda del módulo Task. Esto es así debido a las grandes ventajas que aporta. Por un lado, podemos normalizar la ejecución de tareas con fines dispares, agrupando sus propósitos a través de funciones comunes, tales como, comunicar un evento, solicitar información, etc. Este aspecto será tratado en la parte correspondiente. A modo de resumen, todas las tareas, como parte principal de su existencia, cuenta con un inicio, marcado por el instante del tiempo en el cual debe ejecutarse, así como una condición de fin. Por otro lado, la reutilización de código. Un aspecto realmente importante que incide en el tiempo de desarrollo de las aplicaciones, en este caso, de las tareas. Permite la posibilidad de ahorrar grandes cantidades de tiempo evitando duplicar código ya escrito. Acompañado de unos tests extensos, además, respalda el código con robustez frente a futuros errores y fallos en el sistema.

Verbo	Uri	Nombre	Descripción
GET	/job	job_show	Recuperar un listado de todos los jobs.
GET	/job/:uid	job_detail	Recuperar información detallada del job.
PATCH	/job/:uid	job_update	Actualizar los atributos del job.
PUT	/job	job_create	Crear un nuevo job.
DELETE	/job	job_delete_all	Eliminar el listado completo de jobs.
DELETE	/job/:uid	job_delete	Eliminar un job.

- Logs

Los logs, son la parte de toma de datos de todo el proceso de ejecución que llevan a cabo cada uno de los distintos recursos de la red. Es decir, cada nodo, va provisto de distintos niveles de registro, con diferentes estados, que permiten mostrar todo lo relacionado con la ejecución de la tarea, así como el mismo funcionamiento de la instancia desplegada en el servicio del proveedor. A través de la API, es posible consultar dichos registros, para visualizar el estado de todos y cada uno de ellos. Esta característica es sumamente importante, ya que la computación del sistema es distribuida no entre instancias, sino entre servicios diferentes. Inspeccionar uno a uno resultaría una tarea realmente compleja, en caso de que fallase una, con motivos difusos, podría ser un inicio, para saber realmente dónde está el problema. Pero el objetivo último de esta característica es cerciorarse de que todo funciona correctamente.

Verbo	Uri	Nombre	Descripción
GET	/log	log_show	Recuperar un listado de todos los logs.
GET	/log/:uid	log_detail	Recuperar información detallada del log.
DELETE	/log	log_delete_all	Eliminar el listado completo de logs.
DELETE	/log/:uid	log_delete	Eliminar un log.

- Creación de recursos

Desde el servidor, debe ser posible la configuración de las instancias de las capas inferiores. Esta tarea sólo será ejecutada una vez, cuando se cree la cuenta en el servicio de cloud. Posteriormente, podrá ser ejecutada para reconfigurar algún parámetro. Este paso es crítico, ya que una mala configuración, o la falta de archivos, podría ocasionar graves errores, que podrían ser difíciles de detectar y solucionar. Se supone que las cuentas las crea el usuario manualmente y respetando todos los términos y condiciones que el proveedor exige. No se viola ninguno de ellos. Es por ello que la creación de cuentas queda a cargo del programador. Si bien es cierto que se usarán cuentas gratuitas, siempre se debe respetar las exigencias de la cuenta. Cualquier modo de creación que no sea manual está exento del ámbito de este proyecto.

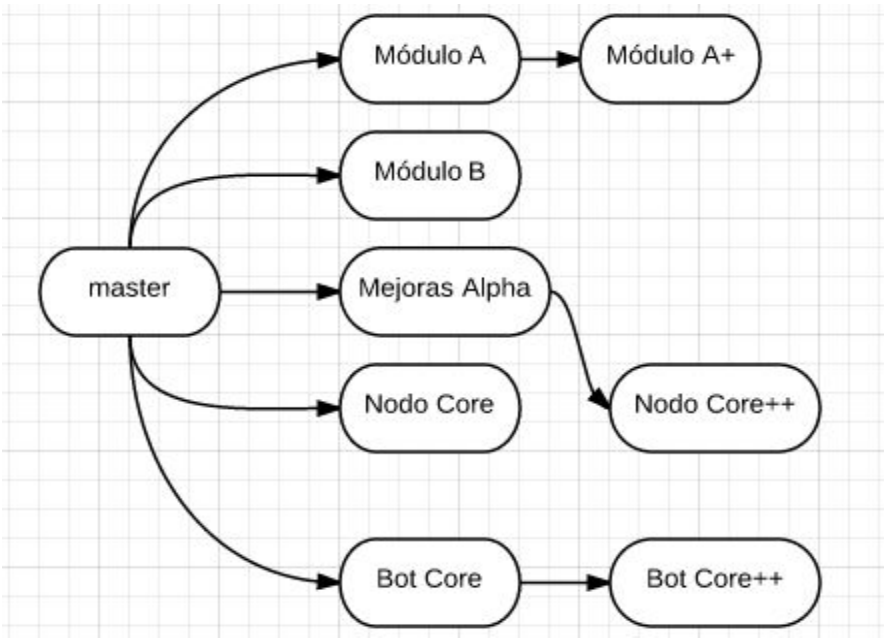
El modo de operación aquí es distinto, puesto que no tenemos un sistema que responda y ejecute las acciones pertinente en consecuencia. Debemos usar otro tipo de técnicas para desplegar la solución en el servicio correspondiente. La automatización de este proceso es muy compleja, ya que cada proveedor propone sus propias herramientas y modos de operación. Para ello, se describe el proceso que se puede seguir. Todos los pasos no serán mostrados, sino que se resumirán y explicaran con mayor extensión en los anexos correspondientes. La siguiente forma, es una entre muchas diversas por las que llegar a tener una solución funcional. Existen diversas herramientas de terceros que tienen operativas distintas, pero todas ellas con la misma o similar finalidad. Ha sido elegido Heroku por su gran popularidad y facilidad de uso, así como Git, el control de versiones más potente que existe actualmente y que goza de una mayor popularidad que ningún otro.

Pasos a seguir(a través de Git y Heroku):

- Creación de un repositorio local en Git
- Creación de un heroku remoto
- Adición de la instancia remota heroku a la configuración Git.
- Pusheo del proyecto a Heroku a través de Git.

<u>Modos de organización</u>

Las posibilidades que nos brinda Git son muy grandes. En este aspecto, podemos proceder de dos modos distintos. Podemos usar una característica muy importante, tal como las ramas. Git permite crear ramas independientes entre sí, con lo cual, podríamos crear un rama para cada capa. Podríamos tener una rama master, para todo el core de nuestra aplicación. Por otro lado, podríamos tener una rama por módulo, así como otra con los módulos básicos y la instancia del recurso correspondiente, nodo o bot. Las posibilidades son infinitas. Es cuestión de organización. Como se visualiza en el gráfico.



Otro alternativa sería crear el sistema a través de tags en Git. Ambos pueden complementarse perfectamente. Combinados nos proveerán de una potente herramienta.

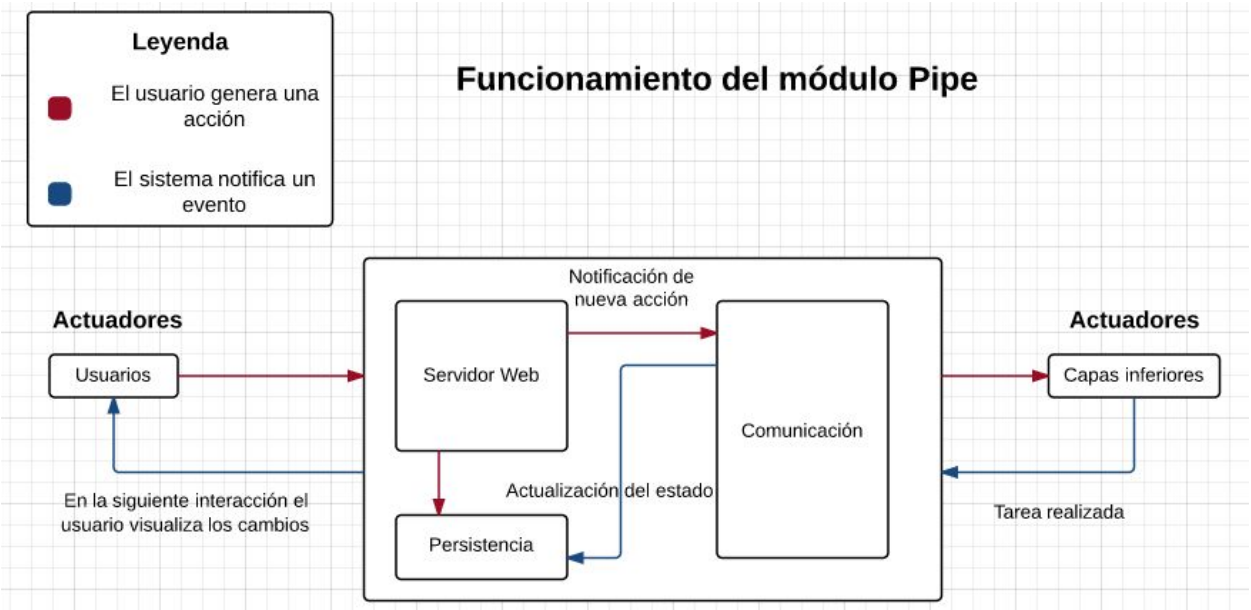
Todo lo expresado anteriormente, puede ser aplicado desde repositorios remotos, así como locales.
A modo de ampliación, sería posible crear un cliente ssh para conectar con las cuentas y poder desplegar los archivos necesarios. También sería posible la utilización de módulos que permitan estos modos de operación(conexión a ssh y ejecución de comandos), pero queda fuera del proyecto actualmente.

Mejoras:
Por cuestiones de complejidad y carga del servidor, este tipo de logs, se solicitan a través de los nodos. Sería posible desarrollar un módulo para solicitarlos directamente a los bots, a través de UDP, cosa que aumentaría en gran medida la velocidad de respuesta. Esta es una mejora que queda en la decisión del programador. Resulta sencillo sustituir la lógica actual e implementar una nueva para especializar las partes del aplicativo. Estas mejoras quedan fuera del ámbito de aplicación del framework actualmente.

Notas acerca de la comunicación:
Como se ha podido observar, la elección del protocolo de comunicación a través de HTTP no es fortuita. Con un rápido vistazo, se puede observar la relación que existe entre el recurso y sus acciones, de una forma limpia y ordenada. Esta es una de las características que tiene HTTP, deja al programador hacer su estructura, el sistema se encarga de todo lo demás. Ha sido posible estandarizar el protocolo, únicamente relacionando las acciones con los verbos del protocolo.

Pipening(canal de comunicación entre procesos) de Comunicación y Servidor Web

La comunicación entre ambas partes, se realiza mediante un módulo encargado de ello, llamado *Pipe.* El funcionamiento de este módulo es sencillo, actuar de pasarela entre ambos extremos, pero con la particularidad invertir el orden de precedencia. Existen dos posibles casuísticas que desembocarán en una prioridad sobre uno u otro. Funcionamiento reflejado en el siguiente gráfico:



En la primera posibilidad, el desencadenante del evento, es el usuario. Cuando el usuario introduce nuevas acciones, el servidor web interpreta la petición del usuario y guarda los campos a través de su capa de persistencia. La comunicación con el otro extremo(Comunicación), podría resultar lenta si utilizamos técnicas de polling. Tratándose de javascript, lo que hará es comunicar al lado opuesto que hay una nueva acción disponible. Este cambio provocará que la parte de comunicación invoque las acciones pertinentes para notificar a las instancias inferiores los nuevos cambios.

En la segunda posibilidad, el desencadenante del evento, es el sistema. En el momento en el cual notifica al servidor que ya ha terminado las tareas. En este caso, el usuario, el actuador del evento en la casuística anterior, no podrá visualizar el cambio hasta que no refresque la página. El módulo de comunicación, utiliza la persistencia del servidor web para actualizar el estado de los recursos.

Rendimiento

En cuanto al rendimiento que podemos alcanzar usando este framework destacan dos vertientes, que no son excluyentes entre sí, sino que combinadas permiten una mayor velocidad. Todas las técnicas siguientes pueden ser aplicadas a la metodología de las capas posteriores. En el caso en que el servidor soporte un gran tráfico de usuarios, o tenga una red de nodos muy amplia, es posible segregar las dos partes que componen el servidor en dos instancias diferentes, tanto en el mismo proveedor como en distintos.

A fin de aumentar el rendimiento del lado del servidor, existe la posibilidad de, nuevamente, crear un módulo dedicado a las descargas de las tareas por parte de los bots. Actualmente, las tareas se envían a los nodos, quienes se encargan de transmitirlo a sus bots correspondientes. Sería posible crear un módulo que sirviera los archivos necesarios, utilizando técnicas de caching, para aumentar la velocidad, o incluso, derivando las descargas a CDN. Aspecto que queda totalmente fuera del ámbito del proyecto.

Persistencia de los datos

La persistencia es un aspecto vital, que permite preservar el estado del sistema en su totalidad, para que el usuario final pueda consultar de todos y cada uno de los aspectos que componen el sistema. Podemos distinguir entre dos tipos de persistencia:

- De código: se refiere a el código que provee de estabilidad a cada una de las partes del sistema en su conjunto. Es decir, todos los archivos que hacen funcionar la instancia desplegada en una máquina. Esto es, por ejemplo, los archivos javascript del bot, junto al árbol de directorios, los módulos pertinentes y los de configuración. Todo el conjunto, es posible almacenarlo en conjunto de un sistema de control de versiones, para realizar las acciones pertinentes nombradas anteriormente.
- De datos: con esto se refiere a el estado de las tareas en su conjunto. En este punto, cabe destacar, que la solución final, aunque resida en el servidor, todos los nodos y sus respectivos bots, contienen trazas de la ejecución de dicha tarea. Es por ello, que el servidor, a través de las operaciones pertinentes, puede consultar datos específicos a todos y cada uno de las partes del sistema. Por otra parte, también se almacenan datos relativos al servidor que nada tienen que ver con las instancias inferiores, tales como, logs de actividad, acciones llevadas a cabo por el usuario, etc.

Vistos los aspectos que componen el sistema la capa de persistencia de los datos, es necesario seleccionar el sistema que los almacenará. Para los archivos no cabe duda, pero para los datos, existen múltiples opciones y variantes que tienen ventajas y desventajas. En este punto, existen dos corrientes, por un lado, las bases de datos relacionales(o SQL). Las cuales, permiten establecer interconexiones o relaciones entre los datos (que están guardados en tablas), y a través de dichas conexiones relacionar los datos de ambas tablas. Por otra lado, las bases de datos no relacionales(o NoSQL) se refieren a este tipo de bases de datos como almacenamiento estructurado, término que abarca también las bases de datos relacionales clásicas. A menudo, las bases de datos NoSQL se clasifican según su forma de almacenar los datos, y comprenden categorías como clave-valor, las implementaciones de BigTable, bases de datos documentales, y Bases de datos orientadas a grafos. La elección de este tipo de tecnología viene marcada por su dinamismo a la hora de almacenar los datos. En el framework, todos y cada uno de los puntos, han sido pensados y desarrollados para sea sencillo alterar su comportamiento de la forma más eficiente posible. Como implementación, podríamos optar a una de las más potentes del mercado si fuese necesario, Redis, pero su coste es alto, debido a su gran agilidad en operaciones NoSQL. Para el framework se ha elegido la implementación MongoDB, debido a su gran popularidad, respaldada por una gran comunidad y una robusta documentación. Existe un web service basado en MongoDB, Mongolab, el cual nos permitirá disponer de bases de datos, sin necesidad de aplicar las extensiones pertinentes en las instancias, con un máximo de 3 de forma totalmente gratuita. Esta opción es muy interesante, porque nos permite consumir los plugins de las instancias en otros aspectos.

Nodos(Controlador)

El nodo es una de las partes críticas del framework. Si bien es cierto, que la carencia de cualquier parte, puede resultar perjudicial para el funcionamiento del sistema, el nodo es el puente de interconexión entre el usuario y la capa de trabajo efectiva. Su cometido es vital, también en el encargado de comunicar el trabajo a los bots bajo su mando. Así como de unir las soluciones que los bot proporcionan, y aunar fuerzas con el resto de nodos para hallar una solución común.
Los nodos son los encargados de comunicar a los bots las tareas a realizar, y proporcionar la información necesaria para que puedan ser llevadas a cabo. Así como, consumir datos que fueran requeridos para completar la tarea. Entre un nodo y un bot, existe un canal de comunicación que permite que sea bidireccional. Para esta tarea es posible usar Websockets(Socket.io), que sirven como wrappers de comunicación a nivel de aplicación. En caso de ser necesario un mayor consumo de información se pueden usar protocolos propios sobre capa transporte(TCP).

Módulos core

Los módulos core, son los referentes al funcionamiento de la instancia desplegada en el servidor. Se encargan de realizar tareas comunes. Estos módulos pueden ser usados por otro capa de la arquitectura, tal como los nodos, ya que comparten el funcionamiento básico. Con esto se pretende evitar la duplicidad del código, el acoplamiento, así como la reusabilidad. Con el desarrollo de módulos simples y eficientes, acompañados de unos test que avalen su efectividad, se consigue una gran calidad en el código escrito.

server
este módulo se encarga de iniciar y gestionar un servidor http. Podemos interpretar las órdenes recibidas del nodo correspondiente y ejecutarlas.

communication
este módulo permite a las tareas comunicar sus resultados con los nodos.

cipher
nos provee de métodos de cifrado y descifrado. En el módulo del servidor, nos permite descifrar los mensajes, mientras que en el módulo de comunicación, nos permite cifrarlos.

Ejecución del app(core)

El archivo encargado de llevar a cabo el buen funcionamiento de la instancia es el llamado app.js, su funcionamiento es el siguiente:

```
importar módulos core;

variable resultado = nada;

funcion gestionarPeticionEntrante():
  variable ruta = detectar_ruta();
  si ruta->tipo == ‘task’:
    comunicar_accion(ruta);
  si ruta->tipo == ‘config’:
    comunicar_accion(ruta);
  sino si ruta->tipo == ‘extra’:
    comunicar_accion(ruta);
  sino si ruta->tipo == ‘servidor’:
    comunicar_accion(ruta);
  sino si ruta->tipo == ‘bot’’:
    comunicar_accion(ruta);
  sino:
    resultado = ‘no encontrado’;
    error = verdadero;
fin funcion;

funcion comunicar_accion(accion):
    enviar_peticion();
fin funcion;

funcion responder(respuesta):
  si error:
    respuesta.escribir(error);
  sino:
    respuesta.enviar(resultado);
```

Comunicación de servidor

Con el módulo server a la escucha, la instancia puede recibir órdenes, y realizar cambios en el comportamiento de la ejecución, según las exigencias del nodo encargado de llevar a cabo este tipo de operaciones. Este proceso se lleva a cabo a través de un pipennig. El servidor(modelo) envía la petición al nodo. Éste, envía la petición al bot, y manda la respuesta al servidor. Este tipo de operación, es el más sencillo de implementar, ya que la respuesta es de tipo bloqueante. En caso de querer acelerar el proceso, es posible hacer las peticiones de una forma aśincrona, es decir, contestar al servidor, mandar pregunta al bot, y cuando éste responda, mandar petición con el resultado al servidor.

- Extra

Este tipo será el encargado de los logs. Puede ser utilizado para aspectos que no afecten al funcionamiento de la instancia. Estos comandos no son ejecutados por el nodo, sino que se manda la orden a el bot correspondiente, y acto seguido se manda la respuesta a el servidor.

Verbo	**Uri**	**Nombre**	**Descripción**
GET	/log	log_show	Recuperar un listado de todos los logs.
GET	/log/:uid	log_detail	Recuperar información detallada del log.
DELETE	/log	log_delete_all	Eliminar el listado completo de logs.
DELETE	/log/:uid	log_delete	Eliminar un log.

- Task

Permite el acceso a las task.

Verbo	**Uri**	**Nombre**	**Descripción**
GET	/task	task_show	Recuperar un listado de todas las task.
GET	/task/:uid	task_detail	Recuperar información detallada de la task.
PATCH	/task/:uid	task_update	Actualizar los atributos de la task.
PUT	/task	task_create	Crear una nueva task.
DELETE	/task	task_delete_all	Eliminar el listado completo de tasks.
DELETE	/task/:uid	task_delete	Eliminar una task.

- Config

A través de esta API, es posible modificar el comportamiento de la instancia desplegada, cambiando los parámetros de configuración.

Verbo	**Uri**	**Nombre**	**Descripción**
GET	/config	config_show	Recuperar un listado de todas las configuraciones.
GET	/config/:uid	config_detail	Recuperar información detallada de la configuración dada.
PATCH	/config/:uid	config_update	Actualizar la configuración.
PUT	/config	config_create	Crear una nueva configuración.
DELETE	/config	config_delete_all	Eliminar todas las configuraciones.
DELETE	/config/:uid	config_delete	Eliminar una configuración.

Comunicación de bot

El nodo está a la escucha de estas acciones que son emitidas por el bot. Acto seguido se efectúan las acciones pertinentes.

Verbo	**Uri**	**Nombre**	**Descripción**
PUT	/task/subset	task_subset_solution	Comunicar los resultados de la task.
PUT	/task/status	taks_status_put	Comunicar el estado de la task.

Comunicación de servidor

Verbo	Uri	Nombre	Descripción
PUT	/task/solve	task_solve	Comunicar la solución de la tarea.
PUT	/task/status	taks_status_put	Comunicar el estado de la task.

Estructura de directorios

/ *raíz de la instancia*
  modules/ *directorio de módulos*
    scheduler.js
    server.js
    cipher.js
    communication.js
   config/ *directorio de configuración*
    routes/
      node.json *archivo de las rutas a las que responderá el servidor.*
      server.json *archivo de las rutas a las que enviará información al servidor*
      bot_task.json *rutas donde enviará información de la task al nodo.*
      config.json *configuración global de la instancia.*

  app.js *archivo core que se ejecutará como archivo principal.*

Bots(Modelo)

El bot, será la instancia más inferior en la jerarquía de la arquitectura. De él depende la realización de las tareas en sí, bajo las condiciones impuestas por el usuario a través del servidor, y transmitidas hacia los nodos, y a su vez, a todos sus bots correspondientes. Su funcionamiento es sencillo, lo máximo posible par poder aprovechar al máximo la capacidad de la máquina en la que está alojada. Los bots están aislados los unos de los otro, así como del server. Solo mantienen comunicación con los nodos.

Módulos core

Los módulos core, son los referentes al funcionamiento de la instancia desplegada en el servidor. Se encargan de realizar tareas comunes. Estos módulos pueden ser usados por otro capa de la arquitectura, tal como los nodos, ya que comparten el funcionamiento básico. Con esto se pretende evitar la duplicidad del código, el acoplamiento, así como la reusabilidad. Con el desarrollo de módulos simples y eficientes, acompañados de unos test que avalen su efectividad, se consigue una gran calidad en el código escrito.

scheduler:
este módulo nos permitirá ejecutar la tarea en un instante determinado. Está provisto de un método principal, que nos permite dejar en espera un tiempo determinado, hasta que se ejecute el callback que le proveemos.

server
este módulo se encarga de iniciar y gestionar un servidor http. Podemos interpretar las órdenes recibidas del nodo correspondiente y ejecutarlas.

communication
este módulo permite a las tareas comunicar sus resultados con los nodos.

cipher
nos provee de métodos de cifrado y descifrado. En el módulo del servidor, nos permite descifrar los mensajes, mientras que en el módulo de comunicación, nos permite cifrarlos.

Módulos de modelos

Este tipo de módulos son los encargados de proveer una interfaz transparente entre los recursos y las instancias. De esta forma, en el caso de las task, con un manager que nos permite ejecutar operaciones CRUD, por ejemplo, podemos crear tareas con una simple función create. De forma transparente y ajena al desarrollador, creará la carpeta con el UID del task, el fichero de ejecución de la task y el de configuración. De la misma forma, para cualquier otro tipo de operación. Esto nos permite desacoplar el código de una forma muy sencilla. Cuando se desee cambiar el modo de almacenamiento, no habrá que modificar la lógica del aplicativo al completo, sino, únicamente la parte encargada de interactuar con el método utilizado para almacenar esos datos.

Todos los módulos implementan los métodos de la metodología CRUD, según sus necesidades. Algunos de ellos implementan el método pero no lo usan, como es el caso de los logs, pues tienen como propósito guardar un estado en un instante determinado, pues, no tiene sentido querer actualizar ese registro.

task
implementa todas las opciones de CRUD.

route
al igual que task, implementa todas los métodos CRUD.

log
solamente implementa los métodos de creación y obtención.

Ejecución del app(core)

El archivo encargado de llevar a cabo el buen funcionamiento de la instancia es el llamado app.js, su funcionamiento es el siguiente:

```
importar módulos core;
importar módulos models;

variable resultado = nada;

funcion gestionarPeticionEntrante():
  variable ruta = detectar_ruta();
  si ruta->tipo == 'task':
    ejecutar_accion(ruta, task_manager);
  si ruta->tipo == 'config':
    ejecutar_accion(ruta, config_manager);
  sino si ruta->tipo == 'extra':
    ejecutar_accion(ruta, log_manger);
  sino:
    resultado = 'no encontrado';
    error = verdadero;
fin funcion;

funcion ejecutar_accion(ruta, manager):
  si ruta->verboHTTP == 'GET':
    manager.read(ruta);
  si ruta->verboHTTP == 'PUT':
    manager.create(ruta);
  sino si ruta->verboHTTP == 'DELETE':
    manager.delete(ruta);
  sino si ruta->verboHTTP == 'PATCH':
    manager.update(ruta);
  sino:
    resultado = 'no encontrado';
    error = verdadero;
fin funcion:

funcion responder(respuesta):
  si error:
    respuesta.escribir(error);
  sino:
    respuesta.enviar(resultado);
```

Comunicación de servidor

Con el módulo server a la escucha, la instancia puede recibir órdenes, y realizar cambios en el comportamiento de la ejecución, según las exigencias del nodo encargado de llevar a cabo este tipo de operaciones.

- Extra

Este tipo será el encargado de los logs. Puede ser utilizado para aspectos que no afecten al funcionamiento de la instancia.

Verbo	**Uri**	**Nombre**	**Descripción**
GET	/log	log_show	Recuperar un listado de todos los logs.
GET	/log/:uid	log_detail	Recuperar información detallada del log.
DELETE	/log	log_delete_all	Eliminar el listado completo de logs.
DELETE	/log/:uid	log_delete	Eliminar un log.

- Task

Permite el acceso a las task.

Verbo	**Uri**	**Nombre**	**Descripción**
GET	/task	task_show	Recuperar un listado de todas las task.
GET	/task/:uid	task_detail	Recuperar información detallada de la task.
PATCH	/task/:uid	task_update	Actualizar los atributos de la task.
PUT	/task	task_create	Crear una nueva task.
DELETE	/task	task_delete_all	Eliminar el listado completo de tasks.
DELETE	/task/:uid	task_delete	Eliminar una task.

- Config

A través de esta API, es posible modificar el comportamiento de la instancia desplegada, cambiando los parámetros de configuración.

Verbo	**Uri**	**Nombre**	**Descripción**
GET	/config	config_show	Recuperar un listado de todas las configuraciones.
GET	/config/:uid	config_detail	Recuperar información detallada de la configuración dada.
PATCH	/config/:uid	config_update	Actualizar la configuración.
PUT	/config	config_create	Crear una nueva configuración.
DELETE	/config	config_delete_all	Eliminar todas las configuraciones.
DELETE	/config/:uid	config_delete	Eliminar una configuración.

Comunicación de cliente

Estas acciones, se invocan a través del módulo correspondiente, comunicación, y serán ejecutadas por las tareas cuando lo precisen. Desde el core, se proporciona acceso a ellas. Permiten que el nodo tenga constancia del estado de la task. También permiten al task solicitar información o comunicar el resultado parcial de la tarea que está realizando.

Verbo	**Uri**	**Nombre**	**Descripción**
PUT	/task/subset	task_subset_solution	Comunicar los resultados de la task.
PUT	/task/status	taks_status_put	Comunicar el estado de la task.

Estructura de directorios

/ *raíz de la instancia*
  modules/ *directorio de módulos*
    scheduler.js
    server.js
    cipher.js
  communication.js
  models/ *directorio de módulos de modelos*
    task.js
    route.js
    log.js
    config.js

  tasks/ *directorio donde residirán las tasks*
  config/ *directorio de configuración*
    routes/
      bot.json *archivo de las rutas a las que responderá el servidor.*
      node_task.json *rutas donde enviará información de la task al nodo.*
    config.json *configuración global de la instancia.*

  app.js *archivo core que se ejecutará como archivo principal.*

Configuración

La configuración de la instancia se almacena en un archivo en formato json. Se carga de forma global a toda la instancia en un archivo llamado config.json, en la ruta /config. A través del módulo server, es posible ejecutar las acciones soportadas sobre el modelo de las configuraciones. A continuación se citan las opciones disponibles que existen por defecto:

<u>security:</u>
- cipher
    - password: contraseña utilizada para cifrar y descifrar los mensajes

<u>node:</u>
- address: dirección que se utilizará para comunicar con el nodo superior.
- port: puerto para dicha comunicación

Objetivos cumplidos

A través de esta práctica se ha conseguido entender el funcionamiento básico del Cloud Computing en sus términos generales. Cada proveedor dispone de sus propias herramientas con un mayor o menor grado de efectividad en circunstancias determinadas bajo contextos determinados. Esta tecnología profiere nuevas formas y metodologías de trabajo a contemplar, que proporcionan altos grados de productividad, a precios económicos. El importe de los proveedores, así como cada uno de ellos no son de relativa importancia para el desarrollo de la práctica, ya que es una question personal. La gran mayoría de ellos cumple con unos estándares de calidad, con mayor o menor grado. Es elección del programador elegir bien cual de ellos utilizar, o para que partes es mejor utilizar unos y otros.

Se ha expuesto de una manera clara y precisa el entorno de trabajo utilizado, herramientas expuestas en esta práctica y todo tipo de utilidades para el programador. Todas ellas facilitan el trabajo del desarrollador, aunque ninguna de ellas es vital para la tarea aquí expuesta.

El lenguaje de programación, ha sido elegido a conciencia según las necesidades del proyecto. Node proporciona una nueva forma de solventar los problemas a través de metodologías complemente diferentes.

Por otro lado, la arquitectura ha sido diseñada de forma práctica, clara y de la forma más extensible posible. Pues ello, propicia que un entorno sea menos intrusivo en tareas críticas y permite el desacoplamiento del código. Conlleva, que pueda evolucionar según las necesidades del desarrollador, o de la misma comunidad. Esta solución propuesta es una de las múltiples opciones que podemos implementar para lograr la explotación de los recursos de las máquinas. Es una solución simple, pero basada en el patrón tan importante en estos tiempos, el conocido como Modelo Vista Controlador. No existe una igualdad absoluta, sino una relación con el concepto arquitectural que desarrolla la idea.

Se ha analizado en una pequeña parte, pero dando una base para el estudio del sistema en su conjunto, del coste que puede conllevar redes de elevadas instancias desplegadas en diferentes sistemas. A partir de este estudio, podemos profundizar en aspectos que van, desde la misma estructura de la red, en su nivel de red(IP) hasta el último aspecto de nuestro proyecto, en este caso, nuestro sistema.

Como último objetivo, y no por ello menos importante, se quieren mostrar las ventajas del OpenSource, que tantas empresas están adoptando. Toda la comunidad, ha dado y seguirá dando todo su conocimiento a todos los que lo precisen de forma totalmente desinteresada.

Complejidad

La idea del proyecto, junto con las tecnologías que precisa para llevar a cabo las tareas requeridas le profiere de una dificultad muy elevada, sobretodo, cuando se trata de computación distribuida, como es el caso. Además, la práctica se ha llevado a cabo con lenguajes y técnicas totalmente desconocidas por el alumno, lo que supone un grado añadido.

El entendimiento de las metodologías, la arquitectura y el diseño del sistema, han dado lugar a una búsqueda intensa de información a través de diversos artículos y libros. La documentación acerca de estos temas, es poca, y generalmente desconocida o mala, no por sus intenciones, sino por el desconocimiento en su totalidad de lo que esta tecnología, así como sus posibles metodologías de aplicación en las que pueden desembocar. El diseño de la arquitectura ha resultado complejo, pero muy gratificante llevarlo a la práctica.

El desarrollo de la última capa, específicamente la referente a la interfaz del usuario supone una carga extra de trabajo, que no ha sido posible desarrollar. Este aspecto es el más costoso en tiempo y esfuerzo debido a la complejidad que supone encontrar la perfecta sincronización entre el usuario y el sistema. Cualquier aspecto debe ser estudiado y diseñado por separado para dotar a la plataforma de una forma de trabajo lo más eficiente posible.

Respecto de la otra parte, la encargada de mandar las órdenes, dada su menor relevancia, no ha sido posible desarrollarla en su totalidad, ya que se ha intentado hacer funcional las partes inferiores del sistema para tener una prueba de concepto funcional. Las capas inferiores gozan de un correcto funcionamiento, con la carencia de unos test unitarios. Este proyecto ha mezclado la investigación con el desarrollo, aspecto que resta tiempo del cómputo global en lo que a aspectos mínimos se merece. No obstante, el core del aplicativo es funcional, y es posible interactuar de una forma básica con él. Con la posibilidad de efectuar pruebas y estudios acerca de los distintos aspectos a mejorar en un futuro trabajo.

Futuro trabajo

Realizar una interfaz con el usuario fluida y eficiente que permita interactuar de la mejor forma con el usuario final. Si bien este es un punto que el desarrollador puede implementar según sus exigencias, es necesario crear una base con la que interactuar de una forma básica con el sistema. Así como, mejorar el sistema del servidor, para llevar al extremo todos las prestaciones de la máquina, segregando las partes explicadas en la práctica.

A medida que el sistema crezca, será necesario proveer de nodos auxiliares al sistema, para poder asegurar que las órdenes llegan en el momento determinado antes de ser ejecutadas. Este proceso, debe ser llevado al estudio y posterior desarrollo para poder llegar a un mínimo de estabilidad en el sistema. Conlleva muchos factores añadidos que pueden ocasionar muchos problemas que deben ser contemplados para llegar a una solución eficaz.

Mejorar el sistema de comunicación entre las capas para lograr un mayor grado de eficiencia. Para ello es posible aumentar el número de nodos que permita difundir la información de una manera ordenada, lo más rápida posible. La adición de nodos auxiliares que sirvan de emisores de broadcast podría ser la solución a este problema, así como la adopción de protocolos de comunicación más rápidos que HTTP. A través de paso de mensajes sobre UDP, es posible conseguir una velocidad mucho mayor a la conseguida tras el desarrollo del proyecto. Esto dota al sistema de una mayor reactividad ante cambios.

El principal cuello de botella del sistema, reside en la complejidad de las operaciones que debe llevar a cabo. Es decir, los nodos deben hallar la solución con las soluciones parciales que reciben de sus bots. Es necesario encontrar un equilibrio entre rendimiento y número de instancias que puede soportar. Este tema precisa de un estudio exhaustivo y muy costoso. Pues es uno de los temas más críticos del sistema. A mayor grado de complejidad de condición de victoria(hallar la solución) mayor será el subespacio en memoria necesario para hallar cada una de las soluciones parciales de las instancias inferiores. Este aspecto es clave, y puede llevar al colapso del sistema.

Ser capaz de exponer de una forma clara y concisa un ejemplo en su totalidad, pues es una estructura muy compleja que puede necesitar de condiciones muy particulares difícilmente replicables en función del desarrollador y de sus necesidades. Ha sido pensado para ser lo más extensible posible, pero la necesidad de crear instancias en distintos servicios, es un factor que puede resultar en una desventaja. El diseño, propicia la ejecución uniforme sin importar la plataforma, debido al uso de estándares que evitan que los sistemas precisen de factores externos para entenderse.

Modificar la metodología utilizada para llevar a cabo la propagación de las acciones. Es decir, el servidor, cuando efectúa una acción, debe ser comunicada a las instancias inferiores. Actualmente, por cuestiones de tiempo y complejidad, las peticiones son bloqueantes cuando pueden serlo, es decir, cuando el servidor emite una acción de tipo get_log, el nodo recibe la petición, la comunica al bot, y envía la respuesta de vuelta al servidor en la misma comunicación. Este tipo de funcionamiento, podría ser modificado para utilizar comunicación no bloqueante, a través de llamadas asíncronas. Realizar la llamada, y cuando esté lista, comunicar a acción al servidor, no tener la conexión abierta esperando respuesta.

Conclusiones

La computación en la nube, o Cloud Computing, permite a un un sistema informático actuar como si de un servicio se tratase. Están diseñados de tal forma, para proveer un conjunto potente de herramientas, que facilitan el desarrollo de aplicaciones con un completo desconocimiento del hardware. Esta abstracción es llevaba a cabo a través de técnicas de virtualización, que convierten todos y cada uno de los periféricos en capas de software capaces de interactuar con ellos a un alto nivel(lenguaje de programación). El concepto extiende en diferentes arquitecturas, clasificadas por el nivel de acceso al sistema, desde SaaS hasta IaaS. El primero, provee de herramientas a través de las cuales es posible la ejecución de código de una forma totalmente autónoma a las condiciones necesarias para ello. Esto es posible, gracias a las instancias con preinstalaciones, que incluyen todo tipo de herramientas. Mientras que, en uno de tipo IaaS, podemos tener acceso al más bajo nivel. Es posible interactuar con el hardware a un nivel muy bajo, en función del proveedor de servicios, incluso en idioma ensamblador. Para este proyecto no se ha usado ninguna cuenta de estas características, ya que, los servicios no son gratuitos. Estos productos suelen ser a medida del desarrollador, y bajo condiciones acordadas en cuanto a recursos.

En este proyecto no se ha estudiado el mundo de los proveedores de servicios por dos factores importantes. Por un lado, la diversidad de ellos. Hay una cantidad ingente de diferentes proveedores. No es la finalidad llegar a profundizar en determinados proveedores, sacando como conclusiones sus ventajas y desventajas. Por el contrario, se pretende estudiar la naturaleza del Cloud Computing, sin importar el proveedor. La elección es en última instancia hecha por el desarrollador, según sus necesidades, gustos, precios o cualquier factor que aprecie como positivo o relevante para elegir dicha empresa. Por otro lado, el mundo cambiante en el que vivimos. Cualquier elección, puede ser errónea bajo diversas condiciones, que afectan al sistema, o la problemática a resolver. Incluso, hay factores que escapan al control del desarrollador, que deben ser meditados y comprendidos para encontrar la solución que se adapte a sus necesidades. Este es un punto importante, que no se debe menospreciar, ya que hacer un despliegue correcto, con determinadas condiciones y diversos factores, en sistemas de grandes dimensiones, con una gran cantidad de instancias puede conllevar errores de cálculo en lo que al tiempo se refiere. Cuando hablamos de cuentas con limitadas, como es el caso, este es un factor realmente importante, que condiciona la estabilidad del sistema y el factor de éxito del mismo.

La arquitectura de implementación del proyecto no es nueva. El patrón modelo-vista-controlador es conocido en el mundo del software por el gran uso del que se hace. En todo tipo de proyectos, es utilizado, sin importar su finalidad o naturaleza del problema. Si bien es cierto, que cada sistema, ha de ser estudiado y encontrar la solución óptima que resuelva dicho problema. En este caso, el patrón, ha servido a modo de guía, para segregar elementos del sistema, de una forma muy eficiente. Todo ello, ha propiciado la reusabilidad de código, como pueden ser los módulos. Éstos son compartidos en todas las

capas del proyecto(Modelo-Vista-Controlador), sin importar el rol que ocupaban en el conjunto del sistema. El proyecto, ha sido dividido en capas, tales como, servidor, nodo y bot. Donde cada una de ellas referencia a las diferentes capas, vista, controlador y modelo. Cada capa puede ser dividida en múltiples subcapas, que representan diferentes rutinas. Pero, todas ellas componen la arquitectura aplicada al sistema MVC.

Se ha realizado trabajo en diferentes ámbitos durante el desarrollo del proyecto, desde la investigación, el diseño o la implementación de una solución. Se ha investigado la arquitectura del Cloud Computing, basada en servicios. Ésta propone distintas formas de ejecución de las problemáticas a través de metodologías nuevas y arquitecturas nuevas. En consecuencia, son antiguas en pensamiento, pero nuevas en aplicación. Es decir, fueron concebidas hace tiempo, pero la tecnología no lo permitía. Tras años de investigación y desarrollo en el campo, aparecieron sus frutos. Actualmente, es usada en muchas aplicaciones como capa de posterior, o backend. Permite lograr un grado de abstracción a diferentes partes de la aplicación. En este caso, se ha visto, cómo es posible comunicar diferentes instancias en diferentes proveedores, a través de estándares, que favorecen el entendimiento entre ambas partes, emisor y receptor. El diseño de la solución, ha sido realizado, según patrones ya diseñados e implementados, con eficacia en anterioridad en diversas aplicaciones. La idea de segregar el sistema en capas, facilita el desarrollo y desacoplamiento del código. Esto pone de manifiesto, una mejor calidad del código. En el ámbito de la implementación, ha sido diseñado para favorecer la extensibilidad de todos y cada uno de sus componentes, para proveer de la máxima flexibilidad posible al desarrollador, con independencia de las demas elementos. El lenguaje, junto con sus módulos, ofrece una gran base para el inicio del desarrollo del proyecto. Aun así, es posible utilizar librerías de terceros, que faciliten el trabajo en determinadas situaciones. En este caso, no se han utilizado, pues existe una gran cantidad de módulos residentes en el core de Node Js. La implementación de las distintas capas, se ha llevado a cabo en su nivel más básico. Para dos capas más inferiores, el nodo y el bot, han sido dotados de lógica básica, que permite la ejecución del conjunto del sistema. En última instancia, la capa referente a la vista, el servidor, no ha podido ser integrada, debido a su gran coste en tiempo. Ya que es necesario desarrollar una interfaz para el usuario rica y elegante a través de diversas librerías, que conlleva mucho esfuerzo llegar a dominarlas por completo. Puesto que este punto, es relevante a la vista de cualquier usuario separado del mundo de la informática. El funcionamiento del sistema en sus capas inferiores, otorga la prueba del funcionamiento. La última capa, es relevante, pero no significativa. Podría utilizarse otro tipo de arquitectura y fragmentarla, para poder ser implementada en otro tipo de dispositivos, tales como smartphones. La gran ventaja de la comunicación del sistema y el uso de estándares, proporciona una interfaz transparente, que puede interactuar con cualquier dispositivo que pueda interpretar mensajes HTTP.

El conjunto de los componentes están en un estado prematuro, aunque pueden ser ejecutados, necesitan de una mayor elaboración. Esta es una base por donde iniciar una investigación más profunda acerca de la arquitectura. El código desarrollado evidencia que es funcional y puede ser portado a otros servicios. Pero es necesario que tenga una mayor elaboración, así como, un conjunto de tests unitarios que prueben el buen funcionamiento en todos los casos posibles. Otros aspectos estructurales, como la comunicación, pueden ser modificados o sustituidos por otros tipos. Este es el caso, por el cual, es posible adoptar soluciones que permitan trabajar en un nivel más bajo, cercano a la pila TCP/IP. La cual permite extraer las mayores velocidades en cuanto a comunicación se refiere. El trabajo para desarrollar este modelo es mucho mayor que utilizar capas del modelo, tales como Aplicación. Pero las ventajas de ésta última deben ser meditadas, puesto que el desarrollo de un modelo de comunicación sobre sockets, UDP o TCP, no es algo trivial. Debe ser estudiado con gran precisión todos y cada uno de los factores, por los cuales, se va a obtener una mejora y si va a ser rentable en términos de tiempo de desarrollo. La capa de vista del cliente, aunque es un factor no condicionante del éxito del conjunto, es algo necesario para poder interactuar de una forma más eficiente. Por otro lado, es de vital importancia, modelar un documento lo más sencillo posible, poniendo en evidencia, la complejidad del sistema, así como, poder afrontarla. Uno de los principales problemas para el desarrollador es la replicación del sistema en perfecto funcionamiento. Todos y cada uno de los factores previamente citados componen las guías para seguir con el desarrollo del sistema.

Una de las finalidades de este proyecto, entre otras, ha sido impulsar el uso del Software Open Source. Desde que Linus Torvalds, empezó la carrera meteórica de esta modalidad, la comunidad no ha hecho más que crecer. La gran mayoría de herramientas utilizadas para llevar a cabo el propósito del proyecto, son todas de éste carácter, a excepción del entorno de programación, que es algo secundario. Su única finalidad, es ayudar en el desarrollo. Esta corriente de pensamiento, en la última década, ha ido tomando fuerza. Prueba de ello, es el sitio que aloja una gran cantidad de código como es Github. Ofrece a sus usuarios la posibilidad de publicar sus herramientas, de forma totalmente gratuita, pudiendo acceder cualquier persona a ellas. Basado en un sistema de control de versiones, como es Git, muy potente y versátil. Con el paso del tiempo, coge una mayor importancia, en todos los lenguajes. Atrás han quedado algunos tan importantes como Google Code o Sourceforge. La tasa de creación de nuevos repositorios, es desenfrenada. La comunidad, evoluciona, y muchas personas están dispuestas a mejorar las herramientas publicadas sin ánimo de lucro, para conseguir una mejora en todos los aspectos del mundo de la informática. Esto la convierte en un ente en continuo cambio, de una gran relevancia, capaz de autorregularse. Prueba de ello, son las publicaciones de empresas privadas, que ofrecen sus herramientas, a través de estos canales de comunicación. Cada vez son las mas empresas que se unen a esta moda, una de las últimas y más importantes, por la gran utilización que se hace de su sistema operativo, Microsoft.

Como colofón final, las necesidades de aprendizaje de todas las fases del proyecto han desembocado en una intensa investigación. Las metodologías de aplicación, la arquitectura, el paradigma que representa NodeJs, todo ello sumado, compone una gran fuente de conocimiento que ha sido estudiada de la mejor forma posible. El último fin no es sino el de aprender e intentar estar preparado para los cambios que siguen. Este proyecto ha representado todo un reto tanto a nivel personal, como profesional. Todo evoluciona muy rápido, y es necesario seguir aprendiendo.

Glosario de términos

Framework: un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.
GitHub: GitHub es una forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git.
Repositorio: almacenamiento de todos los datos.
Git: es un software de control de versiones diseñado por Linus Torvalds.
SPA: Single Page Application.
v8: Motor de javascript de google.
I/O: Input Output
POSIX: acrónimo de Portable Operating System Interface, y X viene de UNIX como seña de identidad de la API.
API: Application Programming Interface.
Callback: una devolución de llamada o retrollamada.
Promise: se usa para computaciones diferidas o asíncronas.
Closure: funciones que manejan variables independientes.
TCP: Transmision Control Protocol.
UDP: User Datagram Protocol.
IP: Internet Protocol.
QoS: Quality of service.
Bloqueante: el proceso queda suspendido hasta que pueda proseguir con la ejecución.
No bloqueante: la continuación del proceso es independiente de su consecución.
Bootstrap: proceso de empaquetado de todos los módulos de sistema.
STDIO: Standard Input Output
Buffer: es un espacio de memoria, en el que se almacenan datos de manera temporal.
DNS: Domain Name Server
URL: Unique resource identifier.
VCS: Version control system.
REST: Representational State Transfer
OO: Orientación a objetos.
RFC: Request for comment.
ARPANET: Advanced Research Projects Agency Network.
IEEE: Instituto de Ingeniería Eléctrica y Electrónica.
MongoDB: es un sistema de base de datos NoSQL orientado a documentos, desarrollado bajo el concepto de código abierto.
TLS/SSL: Transport layer security/ Secure socket layer
HTTPS: HiperText Transfer Protocol Secure
HTTP: HiperText Transfer Protocol
CDN: Content delivery network

Bibliografía

Enlaces web:

http://www.gnu.org/philosophy/open-source-misses-the-point.html
https://www.ietf.org/rfc/rfc768.txt
https://tools.ietf.org/html/rfc793
https://es.wikipedia.org/
-> TCP
-> UDP
-> Cloud Computing
-> MVC
-> Seguridad informática
https://nodejs.org/
-> API
https://thinkster.io/
https://developer.mozilla.org/es/docs/Web/JavaScript
http://www.ontsi.red.es/ontsi/sites/default/files/1-_estudio_cloud_computing_retos_y_oportunidades_vdef.pdf
http://stackoverflow.com/
http://www.2ality.com/2014/09/es6-modules-final.html
https://strongloop.com/strongblog/node-js-callback-hell-promises-generators/
http://www.bennadel.com/blog/2820-function-hoisting-prototype-chains-exports-and-process-nexttick-in-node-js.htm
http://howtonode.org/
http://devcenter.kinvey.com/rest/guides/security
https://www.owasp.org

Libros virtuales:

https://github.com/getify/You-Dont-Know-JS
http://restcookbook.com/

Libros:

Introduccion a Nodejs a través de Koans (pdf)
[O`Reilly] - JavaScript. The Definitive Guide, 6th ed.
Mastering Node.js

ANEXOS

FUENTES
Todos los fuentes del aplicativo pueden ser descargados desde el repositorio de github https://github.com/jmahiques/GearBox.

ESPECIFICACIÓN DE CAMPOS EN VERBOS HTTP

La especificación de todas las rutas, puede ser modificada en su totalidad, a excepción de su nombre, el cual es un id que permite identificar la ruta. Estos cambios, pueden ser llevados a cabo, modificando los archivos de configuración de las rutas, el cual han sido desarrollados con ese propósito. En caso contrario, los módulos han de ser modificados en aquellas partes donde el desarrollador precise de un comportamiento distinto al estandarizado por la metodología REST.

Todas las rutas contienen los mismo campos, lo que hace más simple que podamos realizar cambios.

Servidor

Verbo	**Uri**	**Nombre**	**Campos**
GET	/job	job_show	
GET	/job/:uid	job_detail	
PATCH	/job/:uid	job_update	Campos relativos al job que se quieren actualizar
PUT	/job	job_create	UID: string file: archivo js config: archivo js
DELETE	/job	job_delete_all	
DELETE	/job/:uid	job_delete	UID: string

UID: Identificador único del task.
File: Archivo js ejecutable de la task.
Config: Archivo json de configuración de la task.

Verbo	Uri	Nombre	Descripción
GET	/log	log_show	
GET	/log/:uid	log_detail	UID: string
DELETE	/log	log_delete_all	
DELETE	/log/:uid	log_delete	UID: string

UID: Identificador único del task.

Nodo/Bot

Verbo	Uri	Nombre	Descripción
GET	/log	log_show	
GET	/log/:uid	log_detail	UID: string
DELETE	/log	log_delete_all	Eliminar el listado completo de logs.
DELETE	/log/:uid	log_delete	Eliminar un log.

UID: Identificador único del task.

- Comunicación de cliente

Verbo	Uri	Nombre	Descripción
PUT	/task/subset/:uid	task_subset_solution	UID: string key: string value: string
PUT	/task/status/:uid	taks_status_put	UID: string key: string value: string

UID: Identificador único del task.
key: Clave
value: Valor

- Task

Verbo	**Uri**	**Nombre**	**Descripción**
GET	/task	task_show	
GET	/task/:uid	task_detail	UID: string
PATCH	/task/:uid	task_update	UID: string file: archivo js config: archivo json
PUT	/task	task_create	UID: string file: archivo js config: archivo json
DELETE	/task	task_delete_all	
DELETE	/task/:uid	task_delete	UID: string

UID: Identificador único del task.
File: Archivo js ejecutable de la task.
Config: Archivo json de configuración de la task.

- Config

Verbo	**Uri**	**Nombre**	**Descripción**
GET	/config	config_show	
GET	/config/:uid	config_detail	UID: string
PATCH	/config/:uid	config_update	UID: string key: string value: string
PUT	/config	config_create	UID: string key: string value: string
DELETE	/config	config_delete_all	
DELETE	/config/:uid	config_delete	UID: string

UID: Identificador único del task.
key: Clave
value: Valor

INSTALACIÓN DEL ENTORNO DE TRABAJO

Para proceder con la puesta en marcha del entorno de trabajo, se han usado los siguientes softwares:

- WebStorm (Podemos obtener una licencia por 1 año gracias a la condición de estudiantes, en caso contrario se dispone de un trial de 30 dias)
- NodeJS
- git

Todo el software ha sido instalado bajo la distribución Ubuntu 14.04.3 LTS.

Instalando NodeJS

El intérprete puede ser instalado desde los repositorios de nuestra distribución, pero descargando el código desde la página de Node, podemos compilar por nosotros mismos el intérprete en su última versión.

Para instalarlo desde el repositorio:

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

Si queremos instalarlo por nuestra cuenta:

```
$ cd ~/Descargas
$ tar -xvcf node-v0.12.7.tar.gz
$ cd node-v0.12.7
$ ./configure
$ make
$ sudo make install
```

Tras ejecutar estos comandos, el intérprete estará compilado e instalado en nuestro sistema, así como en el path del terminal de linux, para poder usarlo en cualquier momento.

Instalando git

Para instalarlo, como en el anterior caso, disponemos de ambas opciones:

A través del repositorio:

```
$ sudo apt-get install git
```

A través de sus fuentes:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Ahora ya disponemos de git instalado en nuestro sistema.

Instalando Webstorm
Descargamos el ide de su página web https://www.jetbrains.com/webstorm/, o descargar el comprimido desde la consola:

```
$ wget https://www.jetbrains.com/webstorm/download/download_thanks.jsp?os=linux
$ tar -xvcf
$ cd WebStorm-10.0.4.tar.gz
$ ./bin/phpstorm
```

El ide no requiere de instalación, únicamente ejecutando el archivo, se nos abrirá el entorno. Podemos optar por otros como sublime, pero este nos proporciona una serie de features añadidas muy interesantes, tales como debug, marcado automático de sintaxis, documentación de los módulos core de node, etc.

DESPLIEGUE DE CÓDIGO EN INSTANCIAS CREADAS

La siguiente guía, extraída de la documentación de Heroku, es muy ilustrativa, y muestra el proceso de una forma detallada. Para seguirla, primero deberemos instalar su aplicación, la cual, facilita en gran medida todo el proceso. Como alternativa, ya que Heroku es de tipo PaaS, dispone de acceso a través de ssh, con opciones de configuración avanzadas, más potentes que las que integra la herramienta, pero corremos el riesgo de no dejar la instancia configurada correctamente. Esta herramienta dispone de comandos sencillos, con asistentes que nos proporcionan una ayuda extra a la hora de hacer las configuraciones.

Instalando la aplicación heroku toolbet en Ubuntu(fuente heroku.com)

```
wget -O- https://toolbelt.heroku.com/install-ubuntu.sh | sh
```

Acto seguido, iniciaremos sesión:

```
$ heroku login
Enter your Heroku credentials.
Email: adam@example.com
Password (typing will be hidden):
Authentication successful.
```

En este punto, ya disponemos de la herramienta configurada para operar con ella, con nuestro login, el cual nos permitirá ejecutar acciones sobre nuestra cuenta en la nube.

Creando la aplicación Heroku

Para esto necesitábamos instalar heroku-toolbelt. Con esto podremos manejar la aplicación en Heroku desde la consola de comandos.

Ejecutamos este comando en el directorio principal del proyecto:

```
$ heroku create
```

Nota: Si te dice que el comando no fue encontrado y ya instalaste heroku-toolbelt, sólo necesitas reiniciar la consola.

Hacer deploy de un proyecto en Heroku:

Cuando ejecutes este comando, tu aplicación ya estará publicada en la nube de Heroku, es decir que ya podrás mostrar al mundo entero tu trabajo. ¡Esto es genial!

```
$ git push heroku master
```

Acto seguido, la consola arroja la salida:

```
----> Launching... done, v3
      http://nombre-de-la-app.herokuapp.com/ deployed to Heroku
```

Cambiando en nombre-de-la-app

1. Inicia sesión en tu cuenta de Heroku desde su web.
2. En el Dashboard entra en tu nueva aplicación
3. Ve a la pestaña Settings
4. En el campo de texto Name coloca el nombre que quieras darle a tu aplicación y haces click en el botón Rename
5. Para que el cambio tenga efecto debes avisarle al heroku-toolbelt , entonces debes escribirle lo siguiente en la consola, dentro de la carpeta del proyecto.

```
$ git remote rm heroku
$ heroku git:remote -a nuevoNombre
```

Donde nuevoNombre, es el nuevo nombre que le diste a tu aplicación. Listo recarga y tendrás el nombre que querías.

ARQUITECTURA DE UN ARCHIVO DE RUTAS (JSON)

```
{
   "routes" : [
    {
      "type" : "task",
      "prefix" : "/task",
      "actions" : [
        {
          "VERB": "PUT",
          "uri": "/",
          "name": "task_create"
        },
        {
          "VERB" : "GET",
          "uri"  : "/:id",
          "name" : "task_get"
        },
        {
          "VERB": "GET",
          "uri": "/",
          "name": "task_get_all"
        },
        {
          "VERB": "PATCH",
          "uri": "/:id",
          "name": "task_update"
        },
        {
          "VERB": "DELETE",
          "uri": "/:id",
          "name": "task_delete"
        },
        {
          "VERB": "DELETE",
          "uri": "/",
          "name": "task_delete_all"
        }
      ]
    },
}
```

ARQUITECTURA DE UN ARCHIVO DE CONFIGURACIÓN (JSON)

```
{
        "security" :{
                "cipher" : {
                        "password" : "1Oj&+nn](eKvJ!S"
                }
        },
   "node" : {
     "address": "",
     "port" : ""
   }
}
```