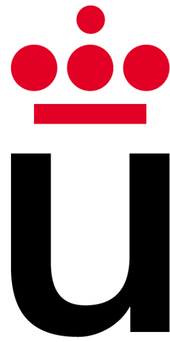

Universidad
Rey Juan Carlos

ESCUELA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER:

Curso académico 2013/2014

ANÁLISIS DE REDES SOCIALES A TRAVÉS DE MINERÍA DE REPOSITORIOS GIT

Autor

Cristian Coré Ramiro Carracedo

Tutor

Gregorio Robles Mártinez

Los grandes logros siempre suceden en el contexto de las grandes expectativas

---

Charles Kettering

Actuar es fácil, pensar es difícil; actuar según se piensa es aún más difícil

---

Johann Wolfgang von Goethe

We try harder

Slogan de Avis

A mi familia. Y a todos aquellos que también me apoyaron.

Índice general

1. Introducción **4**

1.1. Software Libre 4

1.2. Análisis de Redes Sociales (SNA) 6

1.3. SNA aplicado al desarrollo de software 7

2. Objetivos **9**

2.1. Descripción del problema 9

2.2. Alternativas 10

3. Estado del arte **12**

3.1. Enfoques de análisis de comunidades 13

3.2. Mineria de Repositorios de Software 15

3.3. Herramientas 16

3.3.1. Ctags 16

3.3.2. Git 18

3.3.3. Bash 23
3.3.4. Gephi 24
4. Descripción 27
4.1. Análisis de Comunidades 27
4.2. Desarrollo 32
4.2.1. Problemas 32
4.2.2. Solución 35
4.3. Precisión de los resultados 46
5. Casos de estudio 50
5.1. Moodle 50
5.2. OpenStack 59
5.3. Webkit 68
6. Conclusiones 77
6.1. Resultados 77
6.2. Líneas futuras 79
6.3. Limitaciones 79
7. Bibliografía 81

Resumen

En este proyecto nos centramos en el estudio de comunidades de Software Libre, especialmente en el modo en el que los desarrolladores interactúan entre sí. El estudio de estas comunidades y las relaciones entre sus componentes es interesante en un momento en el que el Software Libre es ampliamente utilizado por todo tipo de usuarios (empresas privadas, universidades, instituciones públicas o particulares) y en multitud de áreas diferentes, en las que ya ha demostrado su éxito con creces.

Hay diversos enfoques para representar las interacciones que tienen aquellos que contribuyen a un proyecto de Software Libre. Además de comentar todos ellos, proponemos uno nuevo en el que consideramos que los desarrolladores guardan algún tipo de relación cuando han contribuido en el mismo método o función de un programa de software. Implementamos también un método que considera que los desarrolladores han interactuado entre ellos cuando han modificado alguna vez el mismo fichero. De esta forma tenemos un punto de referencia desde donde empezar nuestro estudio y podemos comparar las diferencias en la representación de una comunidad que produce cada uno de los métodos estudiados.

Por último, mostramos el resultado de la aplicación de ambos métodos a proyectos actuales de Software Libre que poseen una comunidad notablemente grande, sacando las conclusiones oportunas en cuanto al rendimiento de nuestro método.

Summary

In this project we focus on the study of Libre Software communities, especially on how developers interact among themselves. The study of these communities and the relationships among their subjets become interesting in an era where Libre Software is widely used by all sort of users (private companies, universities, public institutions and individuals) and many different areas where it has shown by far its success.

There is a variaty of approaches to represent interactions of those who contribute to Libre Software projects. Besides going through all of them, we propose a new one in which we consider developers to have some kind of relationship when they have contributed to the same method or function in a certain piece of software. On top of that, we implement a mehod where relationships are created as soon as two developers have modified the same source code file. Thus we have a reference point to start our study so that we can compare differences in the representation of the community that our method produces.

Finally, we show the results of applying both methods to current Libre Software projects with large software developer communities. Conclussions will be drawn in order to evaluate the performance of our method.

Capítulo 1

Introducción

1.1. Software Libre

La inherente complejidad del software, junto con otros factores como su eficiencia para resolver problemas o construir soluciones, desemboca frecuentemente en la creación de grandes proyectos. Estos proyectos tienen una función precisa y una serie de características definidas con antelación, siendo capaces de resolver tareas complejas. Como consecuencia, los grandes proyectos son difíciles de controlar y mantener en el tiempo. Éstos son creados tanto por empresas, en las que habitualmente suele ser su producto final, como por comunidades de desarrolladores que no guardan una relación previa.

Durante años, distintas técnicas se han empleado para mantener grandes proyectos de software bajo control, con el objetivo también de aumentar la productividad de los recursos empleados para el desarrollo de éste. De este modo, las empresas invierten un capital considerable en la dirección de los proyectos con el objetivo de alcanzar una estabilidad que permita cumplir unos objetivos de

calidad de los productos desarrollados. Aún así, las técnicas empleadas son en su mayoría empíricas; la hetereogeneidad de los proyectos, así como las diferentes arquitecturas posibles sumado a otras variables como los requisitos (tanto funcionales como no funcionales), el lenguaje de programación o la estructura de la empresa o comunidad entre otros, hacen que la gestión se dificulte y tenga que ser canalizada a través de diferentes metodologías y herramientas para cada caso.

Los proyectos de software privativo, generalmente encontrados en empresas privadas, suelen caracterizarse por tener un alto nivel jerárquico claramente determinado. Se intenta de esta forma tener bajo control el desarrollo del producto a medida que va evolucionando a lo largo del tiempo. Sin embargo, una de las características más importantes de las comunidades de Software Libre es su jerarquía plana [1]. Los desarrolladores y colaboradores de proyectos de Software Libre han demostrado una gran eficiencia en la gestión de éstos sin necesidad de tener un gran control supervisor sobre la comunidad, creando piezas de software estables y ampliamente usadas ahorrando tiempo y recursos en la dirección del proyecto.

Actualmente, muchos de los proyectos de Software Libre son elegidos por encima de cualquier otra alternativa privativa. Este hecho no se debe sólo a la posiblidad de modificar el código para adecuarlo al usuario final, o al bajo coste de adquisición en algunos casos; en muchos campos, los proyectos de Software Libre representan la mejor solución a un problema real, y en algunos de ellos, la solución implementada tiene una calidad tan alta que cualquier otra opción privativa tiene unas posibilidades limitadas a la hora competir por cuota de mercado. Algunos casos destacados son el servidor HTTP Apache, que tiene una cuota de mercado del 61.9 % [2], Wordpress, como gestor de contenidos, con una cuota alrededor del 60 % [3] [4], o GNU/Linux en sistemas operativos para superordenadores, con más de un 90 % [5].

El éxito que ha tenido y sigue teniendo en la actualidad el Software Libre es objeto de estudio. La diferencia en la organización de las comunidades de software plantea una interesante oportunidad de cambio y mejora a aquellas organizaciones que difieren en su *modus operandi*. Tiene sentido pensar que comunidades menos jerárquicas pueden ser más rentables económicamente e incluso tener una organización superior a las comunidades tradicionales de desarrollo de software. Por ello, aumenta el interés en estudiar estos proyectos y sus respectivas organizaciones, para lo que se usan métodos y herramientas disponibles y ya usadas en las ciencias sociales para conocer el comportamiento de ciertos grupos de individuos [6].

1.2. Análisis de Redes Sociales (SNA)

Las siglas SNA hacen referencia a Social Network Analysis. Se trata del estudio de aquellos problemas con forma de red a través del análisis de las relaciones que componen los individuos implicados en tal red. Este problema se caracteriza porque puede representarse como un grafo en el que las aristas son los individuos y los vértices las relaciones entre estos. Dependiendo del tipo de problema del que se trate o del enfoque que se quiera dar a la solución, las relaciones varían en consecuencia.

La rama de la teoría de grafos provee de una serie de conceptos y métodos para el estudio de grafos. Éstos, juntos con otras herramientas analíticas y con métodos para la visualización de redes sociales, componen la metodología de análisis de redes sociales.

A través del estudio de la estructura de una red, es posible sacar conclusiones y encontrar explicaciones al comportamiento de los individuos que conforman la red, e incluso es posible prever el futuro comportamiento de éstas. Por ello,

el análisis de redes sociales tiene una larga historia en las ciencias sociales, aunque otras ramas de la ciencia como las mátematicas, la física, la biología o la informática han colaborado en la mejora de sus métodos de estudio debido a su relevancia en estos campos.

Hay diversas razones para aplicar el análisis de redes sociales, ya que puede aportar mucha información sobre el comportamiento de una comunidad. Entre las más importantes podemos destacar las siguientes:

- Para mejorar la eficiencia de una red de cualquier tipo.
- Para visualizar una red social y encontrar patrones dentro de ésta.
- Para seguir caminos en los que transcurre la información entre los individuos de la red social
- Para realizar un análisis cuantitativo, y en menor medida, cualitativo, con el objetivo de mejorar el entendimiento del funcionamiento de éstas.
- Para realizar un análisis de medios sociales, para mejorar las posibles interacciones de los individuos.

1.3. SNA aplicado al desarrollo de software

Si queremos aplicar el análisis de redes sociales a las comunidades de Software Libre, debemos, en primer lugar, definir qué es una arista y qué es un vértice dentro de un proyecto. Podemos definir un vértice como un desarrollador que ha colaborado en el proyecto; en el momento en el que un desarrollador cambia algún elemento en el código fuente del proyecto, y esto queda registrado en el sistema de control de versiones de tal proyecto, podemos decir que hay una nueva arista en el grafo.

En cambio, la definición de arista puede no estar tan clara en este caso. Esta definición modela la relación existente entre el conjunto de los desarrolladores dentro de un proyecto, por lo que toma una importancia fundamental en la creación de la red y, por consecuencia, del grafo que la representa. Éste es el parámetro que vamos a estudiar para analizar diferentes comunidades desde varios puntos de vista.

Una vez que hemos definido ambos parámetros, el siguiente paso es obtener la información de una red social para poder analizarla. Para ello, hemos de disponer de los recursos necesarios para producir datos legibles. En primer lugar, necesitamos una fuente de datos con la información sobre la red social. En nuestro caso, debido a que analizamos proyectos de Software Libre, tenemos a nuestra disposición el repositorio de código del proyecto. De ahí sacamos la información para poder crear asociaciones entre desarrolladores.

También es fundamental hacer uso de alguna herramienta que sea capaz de operar con estos datos para producir resultados. Esta herramienta debe permitirnos acceder a los datos y poder manipularlos de tal forma que produzca una salida que podamos entender y analizar para sacar las conclusiones oportunas.

Capítulo 2

Objetivos

2.1. Descripción del problema

Teniendo en cuenta que la salida que queremos obtener es una representación de una comunidad de Software Libre lo más acertada posible, hemos de elegir bien cuál es la entrada de datos que vamos a usar para obtener esta información. Una vez tomada la decisión de centrarnos en el código fuente del proyecto en vez de otros recursos más orientados a la comunicación entre desarrolladores, como las listas de correo o los sistemas de seguimiento de errores, es importante detectar correctamente las colaboraciones que han podido tener los participantes del proyecto.

Los sistemas de control de versiones guardan una extensa cantidad de datos, entre ellos, los cambios que producen los desarrolladores en el código, lo cual nos va a ayudar a identificar las conexiones entre desarrolladores. El primer problema al que nos enfrentamos es el de la identificación de información relevante. La forma en la que obtenemos y manipulamos los datos determina las conclusiones

que tomamos después de analizarlos.

Por otro lado, los problemas de visualización de datos y análisis también deben ser solucionados. Dependiendo de las relaciones formadas encontramos comunidades con unas características u otras. En la fase de visualización el objetivo es obtener una representación gráfica que dé una idea visual de cómo está organizada la comunidad de Software Libre. Por último, realizamos un análisis sobre los datos obtenidos, con los cuales calculamos las medidas que usamos para sacar las conclusiones a las que llegamos en cada proyecto.

2.2. Alternativas

En cuanto a reconocer colaboraciones entre desarrolladores, ha de escogerse una implementación para encontrar relaciones de manera fiable, que realmente representen la comunidad que forma el proyecto. Una opción es analizar las líneas de código modificadas por cada desarrollador e intentar encontrar coincidencias entre ellos en vez de hacerlo a nivel de función. Este método resulta más complejo en cuanto a su implementación debido a que un cambio en un fichero afecta a las líneas de código posteriores a éste haciéndolas cambiar de lugar. Por ello, no sería suficiente simplemente con analizar números de líneas de código, sino el código completo, lo que también implica consultas a Git más pesadas, y como consecuencia, más lentas. Además, siendo la definición de subrutina la de pieza de código que implementa o resuelve un problema bien definido, no parece necesario ir más allá en cuanto a granularidad se refiere para asumir una colaboración entre desarrolladores.

Por otro lado, en cuanto a la implementación de la solución, la alternativa principal es el acceso a una base de datos en lugar de acceder directamente al sistema de control de versiones. Para ello, la herramienta CVSAnalY fue la

alternativa considerada, ya que guarda información del repositorio a partir de los *logs* de éste [7]. No obstante, para nuestros propósitos, es necesaria más información que la contenida en los *logs*, por lo que la opción principal pasa a ser el acceso directo al repositorio.

Capítulo 3

Estado del arte

En los análisis de redes sociales hechos con anterioridad, el concepto de relación entre desarrolladores difería del propuesto. En algunos casos, la relación entre desarrolladores es creada en el momento en el que dos de ellos han trabajado en el mismo proyecto [8], [9]. Para ello, se analizan sitios web de colaboración de proyectos de Software Libre como SourceForge [10], desde donde se obtienen los diferentes desarrolladores y los proyectos en los que han trabajado. Una coincidencia de dos desarrolladores en un proyecto crea una relación entre los primeros.

Una forma más afinada de definir una relación entre desarrolladores obliga a conocer los detalles de cada proyecto. Es muy frecuente que éstos estén divididos en diferentes módulos, por lo tanto otra opción es considerar una relación entre desarrolladores cuando ambos han contribuido en el mismo módulo [6], [11]. Esta definición tiene mucho sentido en aquellos proyectos con un número de desarrolladores limitado; por otro lado, en proyectos de gran tamaño es posible que haya una cantidad demasiado alta por cada módulo.

Un paso más en el nivel de refinamiento a la hora de definir una colaboración es usando los ficheros como elementos de unión entre desarrolladores; es decir, detectar los cambios sucedidos en éstos por cada individuo para crear las colaboraciones [12].

También se han dado otros enfoques a la hora de definir relaciones que no tienen tanto que ver con el código del proyecto, sino con las relaciones de individuos de una comunidad en otras plataformas. Por ejemplo, analizando las listas de correos, que pueden ser descargadas para algunos proyectos, se toman aquellos participantes que coinciden en diferentes listas y se asume su colaboración [13]. También se puede analizar el sistema de seguimiento de errores y observar qué sujetos han escrito en las mismas incidencias para crear así los enlaces entre nodos [14].

3.1. Enfoques de análisis de comunidades

Con frecuencia, cuando se ha analizado una comunidad de Software Libre, se ha intentado definir una relación entre dos desarrolladores de una forma aproximada. Es difícil saber con un nivel aceptable de certeza las relaciones que existen entre los diferentes desarrolladores de un proyecto. Uno de los enfoques usados, entre muchos otros, es el de definir una relación cuando dos desarrolladores han trabajado en el mismo fichero del proyecto (han modificado código de tal fichero).

De esta forma se van creando relaciones en la comunidad a medida que los desarrolladores van añadiendo, suprimiendo o modificando código. Cuantos más ficheros hayan manipulado, más probabilidades hay de que tengan relaciones con otros desarrolladores del proyecto, de igual forma que cuanto más haya colaborado un desarrollador en un proyecto, más probable es que se haya relacionado

con otros desarrolladores de la comunidad. A la hora de representar la comunidad en un grafo, si dos desarrolladores han coincidido en un número n de ficheros, entonces el peso de la arista que une a ambos desarrolladores en el grafo es también n.

No obstante, queremos dar un enfoque distinto a la definición de relación entre dos desarrolladores. Es lícito pensar que, aunque dos desarrolladores hayan trabajado en el mismo fichero, se dé la situación de que no hayan tenido contacto. Un fichero de código fuente puede ser tan extenso que es posible que, habiendo dos desarrolladores trabajado en él, no haya relación alguna entre el contenido de sus cambios. Por ello, para aumentar la posibilidad de que dos desarrolladores sean conscientes de la existencia del otro, y por consecuencia haya una relación entre ellos dentro del proyecto, hemos de acotar el ámbito de trabajo que define tal relación.

Cuando un desarrollador se dispone a solucionar un problema en el código, necesita comprender, entre otras cosas, cuál es el funcionamiento esperado, qué pretende hacer el código que provoca el error, y cuál es el problema en su implementación. Con leer el código no suele ser suficiente para entender la ídea que tuvo el desarrollador que implementó la parte del código que ahora falla, y a veces los comentarios en el código no son suficientes. Para poder localizar, entender y solucionar un problema, debe haber comunicación entre el desarrollador encargado de resolverlo y el que implementó dicha solución.

En la gran mayoría de lenguajes de programación, las operaciones complejas o repetitivas con una función definida se abstraen en métodos o funciones. Si definimos una relación entre dos desarrolladores como el momento en el que ambos han trabajado en el mismo método, la posiblidad de que éstos se conozcan (o conozcan de la existencia del otro) es mayor. Por todo esto, hemos de identi-

ficar en qué métodos ha trabajado cada desarrollador para poder relacionarlos después en caso de coincidencia con otros.

Es importante destacar que el peso que tiene una relación entre dos desarrolladores no depende del número de veces en las que éstos han modificado un método o función, sino que es directamente proporcional al número de métodos en los que hayan trabajado en común. Es decir, si dos desarrolladores tiene una sola relación por haber trabajado en un método en común, ésta tiene el mismo peso independientemente de las veces que ese método haya sido modificado por cualquiera de ellos. Esto se debe a que, una vez que se ha encontrado una relación entre dos desarrolladores en un método,

3.2. Mineria de Repositorios de Software

Para obtener toda la información para realizar un análisis, aplicamos minería de datos a los repositorios de código fuente de cada proyecto. Estos sistemas realizan un seguimiento de todas las acciones que se producen en el código fuente del proyecto, siempre y cuando el desarrollador aplique los cambios adecuadamente.

A partir de la información almacenada en el sistema de control de versiones, es posible conocer multitud de datos como, entre otros, qué desarrollador ha cambiado el código, en qué fecha lo ha hecho o los ficheros que ha modificado junto con sus líneas de código. Esta es la información que necesitamos para conocer al detalle las colaboraciones entre desarrolladores, aunque es posible obtener mucha más.

Ya que, además de conocer en que ficheros han contribuido los desarrolladores para encontrar coincidencias entre ellos, también queremos conocer qué métodos de cada fichero han manipulado, tenemos que ir un paso más allá a la hora de rescatar información útil del sistema de control de versiones. Es necesario tener información sobre las líneas de código que han sido modificadas para, sabiendo a qué métodos pertenecen esas líneas, crear una relación entre éstos con el objetivo de comparar después qué desarrolladores han colaborado en los mismos. Para ello usaremos herramientas y técnicas explicadas más adelante (véase capítulo 4).

3.3. Herramientas

3.3.1. Ctags

Ctags, u originalmente *Exuberant Ctags*, en un programa usado para generar etiquetas a partir de un código fuente para facilitar y agilizar la búsqueda de éstas desde otros programas, como por ejemplos editores de texto [15]. El principal uso de esta herramienta en combinación con un editor es el de proveer la función de autocompletar, imprescindible para desarrolladores de software. El número de lenguajes de programación que soporta *Ctags* es, a fecha de noviembre de 2013, mayor de 40, contando por supuesto con los lenguajes más usados para el desarrollo de aplicaciones informáticas. También son muchos los editores de texto y los entornos de desarrollo integrado (IDEs) capaces de incorporar esta utilidad a partir del fichero que genera *Ctags*, como por ejemplo *Vim*, *Emacs* o *UltraEdit*.

A partir de uno o más ficheros de código fuente, *Ctags* es capaz de crear un fichero llamado *tags* que se compone de las etiquetas de los elementos significativos encontrados en el código. La lista de etiquetas generadas es la siguiente:

- Nombres de clase
- Definiciones de macros
- Enumerados
- Definiciones de funciones
- Declaraciones de funciones
- Nombres de *structs*
- *typedef*
- Nombres de uniones
- Variables

El formato de cada etiqueta lo analizamos con un ejemplo real:

DMAX	lib/phpexcel/PHPExcel/Calculation/Database.php	379	f

Podemos distinguir cuatro campos:

- Nombre de etiqueta: coincide con el nombre de la palabra elegida por el desarrollador para definir tal estructura en el código (clase, función, variable, etc.).
- Nombre de fichero: lugar donde se ha encontrado la etiqueta.
- Número de línea: posición de la etiqueta dentro del fichero.
- Tipo de etiqueta: un carácter que define unívocamente el tipo de etiqueta encontrada. En este caso, la letra *f* indica que la etiqueta coincide con un nombre de una función.

El papel que ha desarrollado *Ctags* en el proyecto ha sido la de generar el fichero *tags* que nos permite saber en qué línea se encuentra una función. A partir de este dato, podemos saber si el *commit* realizado por el desarrollador ha modificado o no esa función, lo que nos permite conocer todos los cambios en funciones que ha habido en el código para así poder relacionar a los desarrolladores que han trabajado en las mismas funciones.

Una de las desventajas a las que nos hemos tenido que enfrentar en el uso de *Ctags* ha sido la poca eficiencia al momento de actualizar el fichero *tags* a partir de uno ya creado. Como explica la página de manual del comando *tags*, la implementación del parámetro *-u* de *Ctags*:

> -u update the specified files in the tags file, that is, all references to them are deleted, and the new values are appended to the file. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the tags file.)

En otras palabras, el fichero *tags* se vuelve a generar desde el principio incluso aunque los cambios hayan sido mínimos en el código. Para nuestro propósito, esto es un problema que hemos de abordar debido al considerable tiempo que se tarda en generar este fichero y la cantidad de veces que podríamos estar interesados en ello. Por cada *commit* que analizamos, necesitamos que el fichero *tags* esté actualizado para obtener una precisión óptima en el momento de obtener la función que ha manipulado el desarrollador.

3.3.2. Git

Git es una de las aplicaciones más populares actualmente para el control de versiones, diseñada por Linus Torvalds en el año 2005 y que destaca por su velocidad [16].

Figura 3.1: Logo de Git [17]

Git implementa un sistema de control de versiones distribuido (o DSCM, Distributed Source Code Management), por lo que, a diferencia de otros sistemas centralizados, cada usuario puede tener su copia con plena capacidad de seguimiento con total independencia de acceso a la red, y por ello trabajar en un proyecto en modo *offline* disfrutando de todas las ventajas que Git ofrece.

La cuota de mercado de Git en octubre de 2013 era del 38 %, marcando una subida de 12 puntos en un año y medio, lo que indica un sorprendente cambio de tendencia en el uso de sistemas de control de versiones, siendo Git un aspirante al dominio de estos; no obstante SVN sigue siendo el más usado con una cuota del 46 %, a pesar de sufrir un descenso de 11 puntos en el mismo periodo de tiempo [18].

A día de hoy, muchos proyectos de Software Libre tienen su código fuente en un repositorio Git. Buena parte de responsabilidad la tiene GitHub, una plataforma web para alojar proyectos que usa este sistema de control de versiones, en el que el código, por defecto, se almacena de forma pública pudiendo elegir entre distintas licencias de Software Libre, siendo así accesible por y para la comunidad de desarrolladores. En mayo de 2011, GitHub era la plataforma para albergar código fuente más popular para proyectos de Software Libre, por delante de sitios como Sourceforge o Google Code [19].

Sistema	Licencia	Modelo	Velocidad	Almacenamiento
Git	GNU GPL	Distribuido	Alta	Muy Eficiente
Subversion	Apache	Cliente-Servidor	Media	Poco Eficiente
CVS	GNU GPL	Cliente-Servidor	Media	Poco Eficiente
Mercurial	GNU GPL	Distribuido	Muy Alta	Eficiente

Cuadro 3.1: Sistemas de control de versiones y características [Fuente: elaboración propia]

Entre los sistemas de control de versiones, Git destaca sobre su alta velocidad sobre los demás y su gran rendimiento en proyecto a gran escala. También hace un manejo de ficheros muy eficiente, además de tener un gran repertorio de instrucciones. Además, su capacidad para manejar distintas ramas de código hace que sea elegido sobre otros sistemas, ya que agrega mayor facilidad a la programación en proyectos con muchos desarrolladores. En el cuadro 3.1 podemos ver una comparación con los sistemas de control de versiones más utilizados actualmente [20] [21] [22].

Uno de los conceptos más utilizados en los sistemas de control de versiones es el de *commit*. Un *commit* toma todos los cambios que se han producido respecto a una versión aplicándolos a ésta, creando así una versión posterior. Dicho de otra forma, un *commit* representa un cambio en un proyecto en el cual un sistema de control de versiones realiza un seguimiento. Por ello, es la instrucción que utilizan los desarrolladores para producir cambios en un repositorio. Un *commit* puede consistir en un cambio mínimo en un fichero o en cambios significativos en un gran número de éstos.

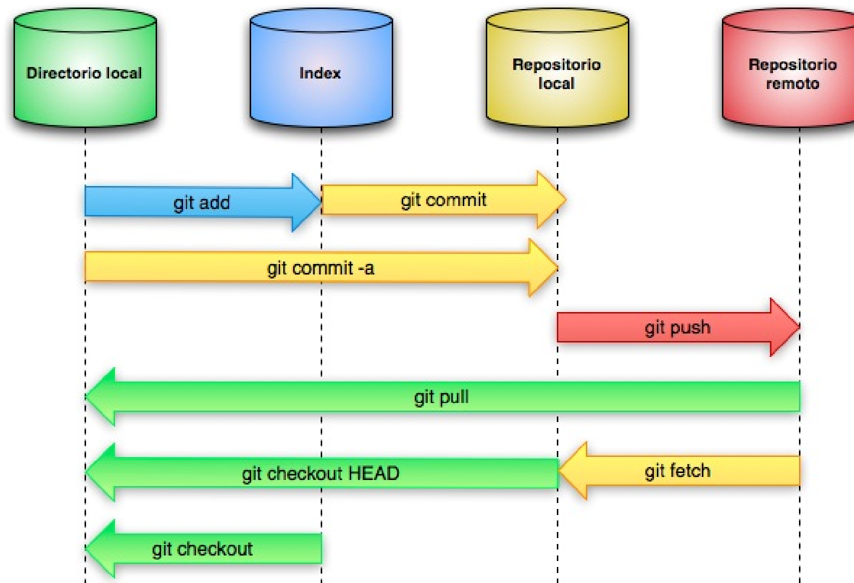

Figura 3.2: Instrucciones en un repositorio Git [Fuente: elaboración propia]

Para facilitar el desarrollo de proyectos en paralelo, los sistemas de control de versiones ofrecen la posibilidad de dividir el código en distintas ramas. De esta forma, varios desarrolladores pueden crear su propia rama para trabajar en grupo o solitario en vez de aplicar los cambios directamente a la rama principal del repositorio. Así se consigue una rama principal más estable, y más tarde es posible fusionar estas dos ramas para aplicar los cambios de una a la otra, resolviendo siempre los conflictos que se hayan podido crear debido a la programación de varios desarrolladores en paralelo. Esta técnica, denominada *merge*, es muy utilizada en grandes proyectos para controlar la estabilidad del producto.

Gracias a la naturaleza del Software Libre, tenemos la libertad de analizar el código fuente. Para ello, no tenemos más que descargar el código a nuestro ordenador personal a través del comando específico para git:

```
git clone [repo]
```

Por ejemplo, para el proyecto Linux usaríamos el siguiente comando:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/mason/linux-btrfs
```

Una vez obtenido el código, un desarrollador perteneciente al proyecto puede modificarlo para resolver un problema o añadir una característica. Después de que el cambio se haya aprobado, se aplica al proyecto para que esos cambios surjan efecto. Por supuesto, Git es quien controla y hace un seguimiento de este cambio.

En el proyecto hacemos uso de algunos comandos de Git que nos facilitan el acceso a los datos del repositorio. Los siguientes comandos han sido utilizados para ello:

- *clone*: Descargar el código fuente del proyecto desde el directorio remoto específicado.
- *log*: Obtener información sobre los *commits* realizados por los desarrolladores.
- *checkout*: Obtener código actualizado a partir de un *commit* de un desarrollador.
- *diff*: Información sobre el cambio realizado en el código de dos versiones de fichero distintas, junto con el código modificado.

Debido a que Git es una sistema distribuido, no es necesario conectarse al repositorio remoto cada vez que queremos obtener información sobre el código,

como por ejemplo la lista de cambios entre versiones o los cambios realizados por un desarrollador. Una vez descargado por primera vez, toda esa información se encuentra en nuestra copia local para poder acceder directamente incluso sin una conexión a Internet. Eso, junto a su rapidez, hacen que sea el sistema ideal para la obtención de datos en grandes repositorios. Para obtener una muestra representativa, hemos de obtener información de un espacio de tiempo que sea suficientemente grande para poder analizar los datos de forma precisa. Además, en muchos casos, tratamos con proyectos extremadamente grandes, por lo que la rapidez en la que Git ejecuta los distintos comandos es fundamental a la hora de determinar el tiempo de obtención de datos.

3.3.3. Bash

Bash es el intérprete de comandos de código abierto instalado por defecto en las distribuciones GNU/Linux [23]. Liberado en 1989, fue programado por Brian Fox para la Free Software Foundation [24] [25].

Además de contener características propias de lenguajes de programación, como por ejemplo estructuras de control o de selección, Bash nos permite ejecutar programas instalados en entornos GNU/Linux, que típicamente están ubicados en el directorio */bin*, */sbin* o */user/bin*, */user/sbin*. Para el manejo de cadenas de texto, utilizamos las siguientes herramientas:

- Perl

  Es en sí mismo un lenguaje de programación, el cual destacada por su manejo de cadenas de texto. Es un lenguaje imperativo derivado de *C*, con características de programación en *Shell*. Perl tiene multitud de funciones y es ampliamente usado para *scripting* y entornos *Web* [26].

- Sed

Sed es un potente procesador de cadenas de texto para Unix. Es útil para manipular cadenas y realizar acciones como cortar, buscar o reemplazar texto. Admite como entrada un fichero de texto que es leído y procesado línea a línea.

- AWK
  Es un lenguaje de programación específico para procesas textos, tanto ficheros como flujos de datos. Destaca por su facilidad para manipular columnas, adaptándose así a la salida de otros programas.

Además, para otras tareas relacionadas con la manipulación de cadenas de texto, también hemos usado programas como *Cut* o *Grep*, con los cuales conseguimos una gran posibilidad de manipulación de textos al aplicar expresiones regulares.

3.3.4. Gephi

Gephi es un software para la visualización y análisis de redes sociales. Esta licenciado bajo GNU GPL y programado en Java, haciendo uso de la librería OpenGL para el renderizado de imágenes 3D [27]. La arquitectura enfocada a la multitarea que se ha empleado en la creación de Gephi permite trabajar con grandes conjuntos de datos y producir resultados visibles claros a la par que elegantes.

Figura 3.3: Logo de Gephi [28]

Gephi desarrolla multitud de utilidades para el estudio y representación de grafos y comunidades, entre otras [29]:

- Análisis de datos: exploración intuitiva para el análisis de datos de redes en tiempo real.
- Análisis de enlaces: revela las estructuras internas existentes entre los elementos del grafo.
- Análisis de redes sociales: fácil creación de conectores de redes sociales para investigar redes sociales.
- Análisis de redes biológicas: representación de patrones de datos biológicos.
- Creación de posters: promoción de trabajos científicos de forma visual con mapas imprimibles en alta calidad.

Los pasos que hemos seguido para la creación de los grafos que representan una comunidad de desarrolladores son los siguientes [30]:

1. Importar conjunto de datos
   La salida del script es el conjunto de datos a analizar, el cual está creado en formato *.csv*. Este formato abierto representa conjuntos de datos de forma sencilla en forma de tabla, simplemente separando los datos en columnas. Las columnas, típicamente, se separan por comas. *Gephi* nos da la posibilidad de introducir un fichero de este tipo como entrada. A partir de éste, reconoce cuántos nodos contiene el grafo y cómo están conectados entre ellos.
2. Disposición del grafo
   Podemos elegir entre diferentes algoritmos para ver la disposición del grafo. Éstos se encargan de reordenar los nodos a partir de sus enlaces en el

conjunto del grafo. De esta forma podemos elegir que los nodos que tienen más conexiones entre sí estén más juntos y visualizar mejor diferentes grupos dentro de una comunidad.

3. Métricas

   Con esta característica podemos calcular las métricas del grafo para realizar un análisis completo de la comunidad.

4. Ranking

   Una vez calculadas las métricas, es posible usarlas para que la representación del grafo dependa de ellas. Así, en nuestro caso particular, colorearamos los nodos dependiendo de su grado (en color más oscuro aquellos con más enlaces) o configurar su tamaño para que coincida con el grado de *Betweenness* (véase apartado 4.1).

5. Diferenciación de grupos

   En aquellas comunidades que haya grupos distinguidos, tenemos la posibilidad de colorear los individuos de cada grupo de igual manera para distinguirlos de otros grupos.

6. Filtrado

   Aquellos nodos que no tienen una gran importancia en la comunidad pueden ser filtrados por distintos parámetros, como por ejemplo su grado.

7. Vista previa y exportación

   Una vez terminado el grafo, podemos pasar a la vista previa para hacer los últimos retoques y, una vez finalizado, exportar el grafo en diferentes formatos.

Capítulo 4

Descripción

4.1. Análisis de Comunidades

Para analizar en detalle una comunidad, creamos un grafo de desarrolladores que están relacionados entre ellos según el parámetro elegido. Este grafo nos da una idea aproximada de cómo es la comunidad que queremos estudiar. Para conseguir un análisis más objetivo, hemos de fijarnos en multitud de parámetros. Antes de todo, definimos los conceptos básicos de un grafo:

Vértice

También llamado nodo, es la unidad fundamental de la que se compone un grafo. Tiene vínculos con otros nodos, y su existencia implica una relación de colaboración en el contexto del grafo.

Arista

También llamada enlace, la arista es la relación existente entre dos vértices de un grafo. En otras palabras, cuando dos nodos están unidos a través de una

arista, existe un vínculo entre ellos.

Grafo

Un grafo se define como un par $G = (V,E)$, donde V es un conjunto de vértices y E un conjunto de aristas que relacionan a tales vértices.

Ya que nuestra forma de representar una comunidad es a partir de un grafo, podemos asignar los conceptos correspondientes:

- Vértice - Desarrollador: Cada nodo del grafo se corresponde con un desarrollador del proyecto.
- Arista - Relación entre desarrolladores: Las aristas representan una relación entre desarrolladores; ambos desarrolladores han trabajado en el mismo fichero o en la misma función, dependiendo del método de estudio que se utilice en cada caso.
- Grafo - Comunidad: El conjunto de desarrolladores y sus relaciones,las cuales son representadas a través de las aristas del grafo, conforman una comunidad.

Como ya hemos comentado, el grafo de nodos y enlaces puede dar una idea aproximada, de forma visual, de las relaciones existentes entre los diferentes miembros de una comunidad, pero para obtener una información más fiable, debemos confiar en parámetros más objetivos. Éstos son calculados a partir de las características de la comunidad. El número de vértices, el número de aristas y la forma en la que los primeros están conectados con los segundos son la base a partir de la cual el resto de parámetros son calculados. Una vez aplicados estos parámetros, que definimos a continuación junto con su significado aplicado a una comunidad, podemos confiar en datos objetivos para analizar diferentes comunidades.

Orden

El orden de un grafo es su número total de vértices, $|V|$. Dado que cada vértice del grafo coincide con un desarrollador, el orden del grafo corresponde con el número total de desarrolladores de una comunidad. De esta forma podemos valorar cuán grande es la comunidad que estamos analizando.

Grado

El grado de un vértice del grafo es el número total de aristas que inciden sobre tal vértice. El grado representa el número de relaciones de un desarrollador con el resto de desarrolladores. Si uno de ellos tiene un grado alto, significa que está muy relacionado dentro de la comunidad.

Grado medio

El grado medio, el cual es un parámetro del grafo completo, es la media de grado de todos los vértices del grafo. El grado medio del grafo nos permite conocer cuán relacionados están los desarrolladores en una cierta comunidad. Si los desarrolladores trabajan en común con frecuencia, se obtiene un valor significativo.

Diámetro

Para aclarar el concepto de diámetro, primero hemos de saber cómo se calcula la distancia entre dos vértices; ésta es el número mínimo de aristas que separa a ambos. El diámetro se define pues como la mayor distancia posible entre dos nodos del grafo. El diámetro del grafo nos da una ligera idea de la dispersión de la comunidad.

Modularidad

La modularidad mide la fuerza de división de una red en diferentes grupos; cuanto más alta sea la modularidad de un grafo, mayor es la conexión entre

diferentes grupos de éste, pero menor es la conexión entre los nodos que conectan los diferentes grupos. Este parámetro nos permite distinguir diferentes tipos de grupos de individuos que se relacionan más entre sí. La modularidad dentro de una comunidad es un parámetro que indica la probabilidad de que haya grupos dentro de ésta, en la que un grupo es un conjunto de nodos muy que se relacionan con frecuencia entre sí. Por lo tanto, en proyectos en los que los módulos de software están claramente diferenciados y sólo ciertos desarrolladores trabajan en ellos, obtenemos una modularidad elevada y un alto número de grupos dentro de la comunidad.

Densidad

La densidad es un valor entre 0 y 1 que mide cuán completo es un grafo, siendo un grafo completo aquel que tiene todos los vértices unidos entre sí. Al tener en cuenta a todos los desarrolladores al momento de calcular la densidad, este parámetro nos permite conocer cuánto colaboran los desarrolladores entre ellos en el conjunto de la comunidad.

Coeficiente de Clustering

Este coeficiente mide cuán conectados están los vecinos de un nodo. Nos permite tener un conocimiento sobre la dispersión de tal nodo. A partir de este parámetro podemos deducir cuán bien conectado está un nodo; aquellos desarrolladores que tienen relaciones con desarrolladores importantes tienen más oportunidades de tener un mayor protagonismo o de estar en un nivel más alto en comunidades muy jerárquicas.

Betweenness

Mide la centralidad de un nodo del grafo, o en otras palabras, la importancia del nodo dentro de tal grafo. Se calcula a partir de la división del número de caminos más cortos que pasan por ese nodo entre el número total de caminos del

grafo. Podemos conocer la importancia de un desarrollador a partir de su grado de *Betweenness*. Un nivel alto significa que la aportación del desarrollador a la comunidad es relevante. Debido a la naturaleza de un proyecto de software, esto significa además que tiene un amplio conocimiento del proyecto y que tiene una responsabilidad destacada dentro de la comunidad.

Longitud media de camino

Este parámetro mide el número medio de saltos entre aristas que un vértice necesita para llegar hasta otro vértice del grafo. Para obtener la longitud media de camino de todo el grafo, calculamos su media a partir del resultado obtenido de cada nodo, teniendo en cuenta que la longitud de camino entre dos vértices unidos por una arista es 1. Nos permite conocer el grado de expansión de la comunidad. En una comunidad con todos los nodos conectados entre sí, el valor sería uno. Por lo tanto, cuantas menos conexiones tenga la comunidad y más expandidas estén dentro del grafo, mayor es este valor, mostrándonos así una comunidad en la que sus desarrolladores tienen pocas probabilidades de conocerse entre sí o de colaborar en el futuro.

Hemos de tener en cuenta que, en nuestro caso, tratamos con grafos no dirigidos. A diferencia de los grados dirigidos, éstos no tienen un dirección específica en las relaciones entre los nodos. Esto se debe a la forma en la que consideramos que existe una relación: cuando dos desarrolladores han trabajado en un elemento en común. Esta asociación no tiene una dirección definida. Por ello, no aplicamos algoritmos conocidos para el estudio de comunidad que si se pueden representar como grafos dirigidos, como por ejemplo el algoritmo HITS, el cual da un peso a los vértices determinando la calidad de sí mismo y de sus vínculos [31], o el algoritmo PageRank, el cual considera la importancia de un nodo según los nodos que le enlacen [32], famoso por ser usado en el motor de búsqueda de Google.

4.2. Desarrollo

4.2.1. Problemas

Antes de llegar a la solución final, se han manejado varias propuestas que estaban destinadas a abordar el problema del acceso y manejo de grandes cantidades de datos. Todas ellas han tenido una o varias razones para ser descartadas, pero las principales han sido un tiempo de ejecución demasiado alto y una extrema complejidad en la manipulación de datos.

En primer lugar, la solución propuesta para el acceso de datos a distintos repositorios era utilizar un paso intermedio en vez de tratar directamente con el sistema de control de versiones. Para ello, se utilizaría la herramienta *CVSAnalY*, la cual extrae información de los logs del repositorio y lo guarda en una base de datos [7], [33].

De esta forma, la obtención de datos se haría a través de consultas a la base de datos creada por *CVSAnalY*. Además de los problemas comunes al hacer grandes consultas a bases de datos (problemas de memoria, manejo de filas borradas, complejidad en el manejo de grandes estructuras de datos, etc.), la principal desventaja de *CVSAnalY* viene dada por el hecho de que no almacena el código que se ha modificado en el repositorio.

Una vez descartada la opción de acceder a los datos del repositorio a través de una base de datos intermedia, hemos optado por acceder directamente al repositorio del proyecto para obtener la información que necesitamos para crear la comunidad de desarrolladores. La principal desventaja de esta solución es el tiempo de ejecución: no obstante, al usar un sistema de control de versiones distribuido no es necesario hacer ninguna consulta remota más allá de descargar

el proyecto, acción que sólo hemos de realizar una vez por cada proyecto.

Una vez que hemos tomado la decisión de realizar consultas directamente al repositorio del proyecto, tenemos que saber qué ficheros y qué métodos han sido manipulados por los desarrolladores. Es un problema trivial conocer los ficheros modificados (un simple comando de Git nos da tal información), pero no lo es tanto para los métodos. Para ello, hacemos uso *Ctags* que, como ya hemos explicado (véase apartado 3.3.1), resuelve el problema de identificación de métodos dentro de un fichero.

El objetivo esta vez es reducir el tiempo de ejecución al mínimo, siempre y cuando la fiabilidad de los datos no quede comprometida. Para ello, cada vez que hay un cambio en el código, hemos de ser conscientes de que el fichero creado por *Ctags* ya no está actualizado; algunos métodos pueden haber cambiado de línea al añadir o quitar sentencias en el código. Por esta causa, cada vez que queremos conocer en qué método se ha realizado un cambio, necesitamos una versión actualizada del fichero *Ctags* para conseguir una precisión óptima. A este problema se suma la falta de optimización en la actualización de tal fichero por parte de *Ctags* como ya hemos explicado anteriormente (véase apartado 3.3.1), lo que incrementa considerablemente el tiempo de ejecución.

Éstas fueron las propuestas iniciales para resolver esta situación intentando ahorrar tiempo sin perder precisión.

1. No actualizar el fichero de *Ctags*
   En este caso, el análisis del repositorio se hace de forma rápida, pero se obtienen resultados poco precisos por la falta de actualización de las líneas en donde se encuentran los métodos a buscar.

2. Actualizar el fichero de *Ctags* por cada *commit*

Actualizando el fichero *Ctags* cada *n commits* podemos conseguir una precisión alta con un tiempo de ejecución significantemente más bajo comparado con la opción de actualizar siempre. Con este método no se alcanza una precisión absoluta.

3. Actualizar el fichero de *Ctags* cada *n commits*

   Al actualizar siempre el fichero *Ctags*, el tiempo de ejecución resulta inadmisible aunque logremos una precisión óptima.

4. Actualizar el fichero de *Ctags* manualmente

   En este método somos nosotros los que calculamos, a partir de la salida del comando *git diff*, que ficheros han sido modificados y cuales son los números de línea de cada método después del cambio. De esta forma se obtiene una alta precisión con un tiempo aceptable. Cabe resaltar que ésta es la opción más compleja de implementar, por lo que se perdería tiempo al calcular los nuevos números de línea de cada método.

5. Actualizar el fichero de *Ctags* manualmente de ficheros modificados

   En este caso, a partir del fichero que *Ctags* ha creado, se actualizan aquellos ficheros que han sido modificados en cada *commit*. Esto se hace ejecutando *Ctags* con el nombre del fichero como argumento. Una vez que se tienen las etiquetas del fichero actualizadas, se pueden borrar las que contenía el fichero *Ctags* creado originalmente y añadir aquellas actualizadas. Estamos contando con la ventaja de que, en cada *commit*, conocemos que ficheros han sido modificados, mientras que *Ctags* desconoce esta información y por ello ha de recorrer todos los archivos de ficheros de nuevo en busca de etiquetas. Con este método se obtiene un tiempo de ejecución aceptable y una precisión óptima.

4.2.2. Solución

Con el objetivo de obtener unas relaciones entre desarrolladores más realistas en las que las probabilidades de que dos sujetos de una comunidad hayan tenido contacto de alguna forma, y por lo tanto podamos considerar que se conocen, refinamos al máximo el concepto de relación hasta definirlo como la colaboración de dos desarrolladores en el mismo método. Así eliminamos la posibilidad de que dos desarrolladores hayan actuado sobre un fichero de gran tamaño, en el que se resuelven múltiples problemas, y no hayan interactuado entre ellos. Al ser el método o función el componente comúnmente usado para encapsular un problema bien definido en la vasta mayoría de los lenguajes de programación, podemos decir que dos personas han trabajado en el mismo problema si ambos han colaborado en el mismo método.

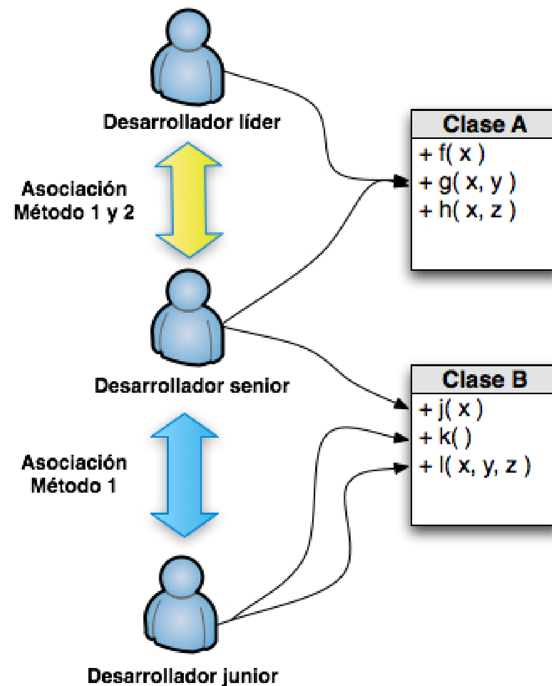

Figura 4.1: Asociación de desarrolladores [Fuente: elaboración propia]

Como vemos en la figura 4.1, dos desarrolladores tienen una relación cuando han colaboran en el mismo fichero de acuerdo al primer método (coincidencia en ficheros), como pasa con el *Desarrollador senior* y el *Desarrollador junior*. En cuanto al segundo método, el cual desarrollamos más adelante, la relación sólo se produce cuando han colaborado en el mismo método de un fichero (*Desarrollador líder* y *Desarrollador senior*).

Además de crear comunidades de distintos proyectos a partir de tal concepto de colaboración, otro objetivo es la comparación con la técnica que define la colaboración entre desarrolladores a nivel de fichero. De este modo, podemos incluir la nueva metodología en el marco de aquellas ya estudiadas con anterioridad y analizar sus pros y sus contras como método de representación de una comunidad.

Para llegar a esta solución, la implementación se basa en un *script* programado en Bash que hace consultas al repositorio Git del proyecto para obtener información sobre las colaboraciones de los desarrolladores al proyecto. Una vez editados estos datos con distintas herramientas para el manejo de cadenas de caracteres, se comparan con el fichero que crea el programa *Ctags* en donde se definen las posiciones de los métodos y funciones en los distintos ficheros del proyecto. Una vez recopilada toda esta información, sólo queda ver qué desarrolladores han colaborado entre sí para crear una salida que nos permita representa la comunidad.

La figura 4.2 nos da una idea de dónde se sitúa cada componente en la solución propuesta. El *script* de Bash hace consultas al repositorio local para producir una salida en la que representamos los datos de las asociaciones entre desarrolladores.

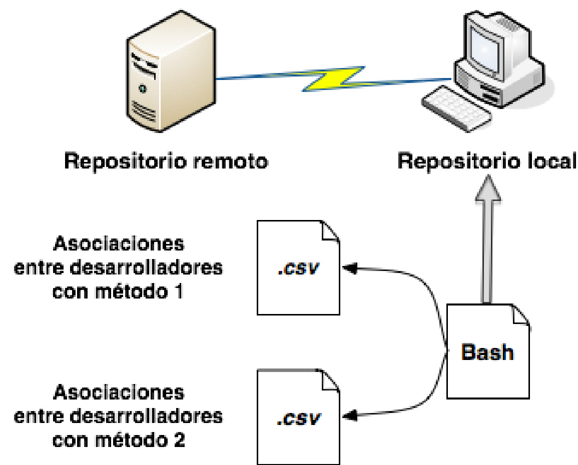

Figura 4.2: Solución propuesta [Fuente: elaboración propia]

Los argumentos que toma de entrada el *script* son los siguientes:

- -h

  Ayuda, imprime la forma en la que se debe ejecutar el *script* con una explicación del significado de sus argumentos y el uso de éstos.

- -f

  Fecha de comienzo desde la que se quiere obtener los datos a analizar. Si se omite, se recopilan datos desde la fecha de comienzo del proyecto. El formato de la fechas es *AAAA-MM-DD*, siendo *A* el año, *M* el mes en formato numérico y *D* el día del mes.

- -t

  Fecha final hasta la que se quiere obtener los datos a analizar. Si se omite, se recopilan datos hasta la fecha actual. El formato es el mismo que el de la fecha de comienzo.

- -r

  Enlace en el que se encuentra el proyecto a analizar (argumento al comando

git clone). Si se omite, se considera que se está ejecutando el script desde el directorio del proyecto (ya ha sido descargado o es un proyecto local).

- -v

  Verbose mode, para obtener información sobre la ejecución del programa.

Una vez se tienen los argumentos, el primer paso es obtener el código. El *script* puede ser ejecutado dentro de un repositorio ya existente o ser descargado por el *script* usando el flag -r, simplemente pasando como argumento la dirección donde se encuentra.

A partir de este momento podemos pasar a analizar el código del proyecto elegido usando Git. Podemos diferenciar el análisis de los resultados en tres pasos distintos:

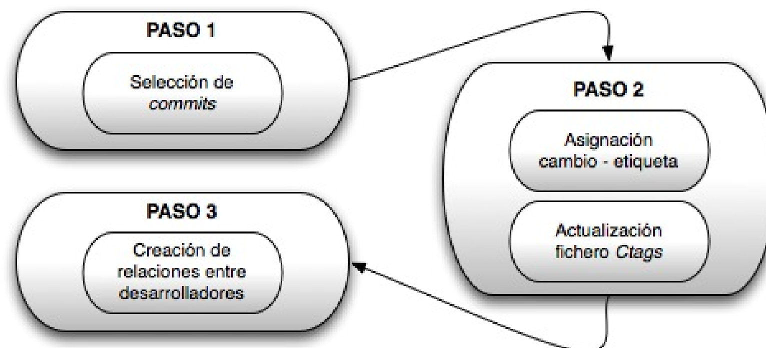

Figura 4.3: Diagrama de flujo de *script* [Fuente: elaboración propia]

1. Elección de *commits* a analizar

Para ello, se pueden utilizar los parámetros *-f -u* para definir las fechas en las que se quiere hacer el análisis. Si no se definen argumentos, se analiza el proyecto completo. Para obtener los *commits* que se han hecho entre unas determinadas fechas, se hace una simple consulta a Git. Esta consulta da como resultado la codificación *hash* de cada *commit* junto con el nombre del desarrollador responsable del cambio. Es necesario que conozcamos quién ha sido el desarrollador que ha creado ese cambio en el repositorio para luego encontrar conexiones entre éstos.

El comando utilizado ha sido el siguiente:

```
git log –all –since="$since" –until="$until" –pretty=format:"%H %an"
```

donde los *flags –since* y *–until* definen las fechas entre las que se quieren obtener los *commits* y el flag *–pretty* imprime la codificación *hash* y el nombre del autor, que corresponden a los parámetros *%H* y *%an* respectivamente.

2. Asignación de cambios a etiquetas

En este paso identificamos qué métodos han sido modificados por cada *commit*. Para ello, nos ayudamos de la cabecera de la salida del comando *git diff*. Este comando compara versiones distintas y nos muestra las diferencias. En nuestro caso, comparamos los *commits* con la versión anterior para saber qué líneas se han manipulado.

Vamos a analizar a partir de diagramas de flujo los pasos que han sido necesarios para lograrlo.

Como vemos en la Figura 4.4, el primer paso es descargar la primera versión del código desde donde queremos realizar en análisis. Una vez hecho, creamos el fichero *d*eCtags desde ese punto. Tal fichero es actualizado

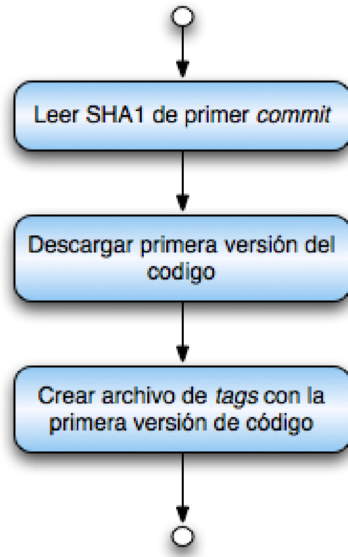

Figura 4.4: Obteniendo código y *tags* [Fuente: elaboración propia]

con cada iteración para obtener información precisa sobre la localización de los métodos en el código como veremos más tarde:

El siguiente paso es, para cada *commit*, identificar ficheros y métodos que han sido modificados y actualizar el fichero de *tags*:

En este paso hemos hecho referencia a dos métodos que analizamos más adelante. Lo que se pretende conseguir es, a partir de las diferencias mostradas de un *commit* a otro, iterar para ir obteniendo el número de línea y el fichero modificado. Recordemos que un *commit* puede haber modificado multitud de ficheros o uno sólo en diferentes líneas. Para obtener toda la información necesaria hacemos uso del siguiente comando:

```
git diff –unified=0 $rev^ !)
```

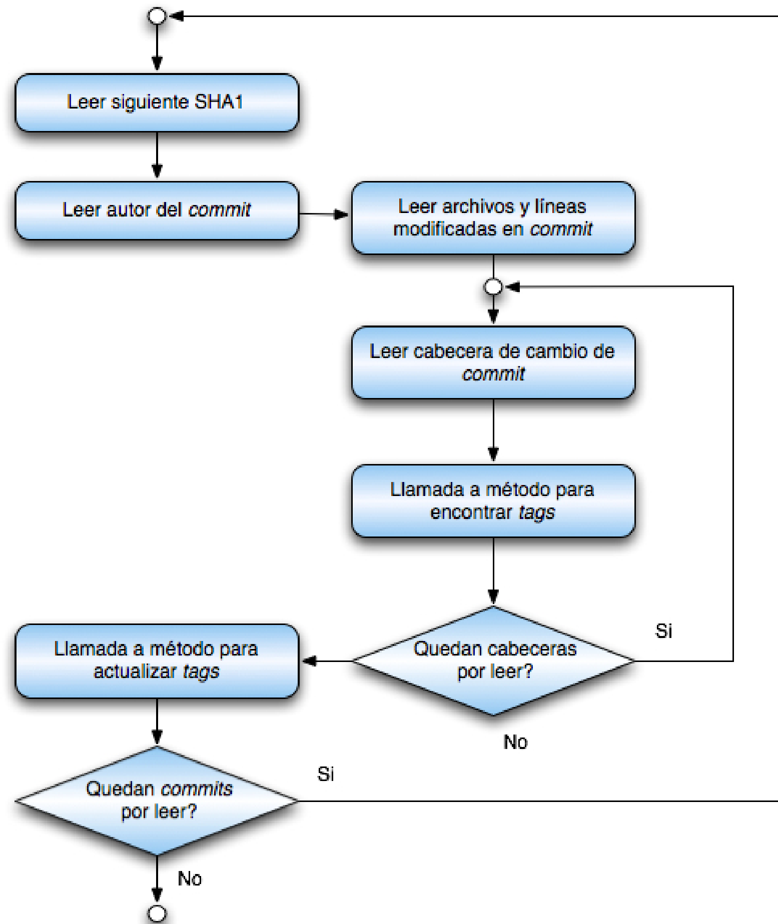

Figura 4.5: Identificación de métodos y ficheros modificados [Fuente: elaboración propia]

Usamos el *flag –unified* para obtener una salida más reducida, ya que el código no nos interesa tanto, sino simplemente la cabecera con la información sobre los cambios. Además, pasamos como argumento *$rev^ !* para hacer saber a Git que queremos comparar con la versión anterior de cada fichero.

```
diff --cc GraphDataCreator.sh
index 5acb759,01a9dea..0000000
--- a/GraphDataCreator.sh
+++ b/GraphDataCreator.sh
@@@ -108,0 -104,0 +108,22 @@@ function findFittingTag(
```

Figura 4.6: Cabecera de la salida del comando *git diff* [Fuente: elaboración propia]

Una vez que tenemos los números en los que se han producido los cambios, los comparamos con las etiquetas producidas por *Ctags*. A partir de ellos podemos concluir qué métodos han sido modificados. Este paso se realiza en el método *findFittingTag*, que mostramos en la Figura 4.7.

Básicamente, se realiza una comparación entre los ficheros modificados de cada *commit* y los *tags* encontrados por *Ctags*. Si no se encuentra un *tag* para un cambio, simplemente se añade como una modificación en el fichero. En cambio, si se encuentran tags, se busca aquella que coincide con más precisión al cambio realizado en el código. Toda esta información se almacena en un fichero intermedio que es leído posteriormente para encontrar las asociaciones entre desarrolladores.

Después de encontrar todas las coincidencias de un *commit* con los números de línea de los métodos calculados por *Ctags*, hemos de actualizar el fichero *Ctags* antes de consultar el siguiente *commit*. De otro modo, estaríamos comparando con unos datos desactualizados. Para actualizar tal fichero, hemos usado el quinto método ya explicado (véase apartado 4.2.1), debido a su aceptable tiempo de ejecución y precisión óptima. Como podemos ver en la Figura 4.8, vamos iterando entre los ficheros modificados y actualizándolos uno a uno para anádirlos al fichero de *tags* del proyecto. Una vez obtenidas las etiquetas actualizadas, podemos analizar el siguiente *commit*, y así sucesivamente.

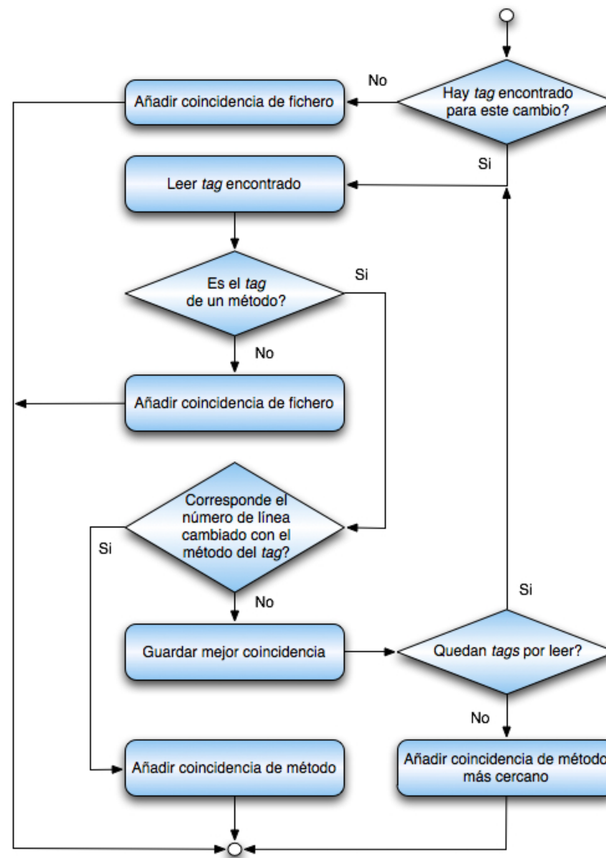

Figura 4.7: Obteniendo coincidencias [Fuente: elaboración propia]

3. Asignación de asociaciones entre desarrolladores

En este último paso se usa la salida del paso anterior para obtener las relaciones entre desarrolladores. De esta forma, una relación se obtiene cuando dos coincidencias método-*tag* se han producido sobre el mismo fichero o cuando se han producido sobre el mismo método, dependiendo del análisis que se quiera hacer.

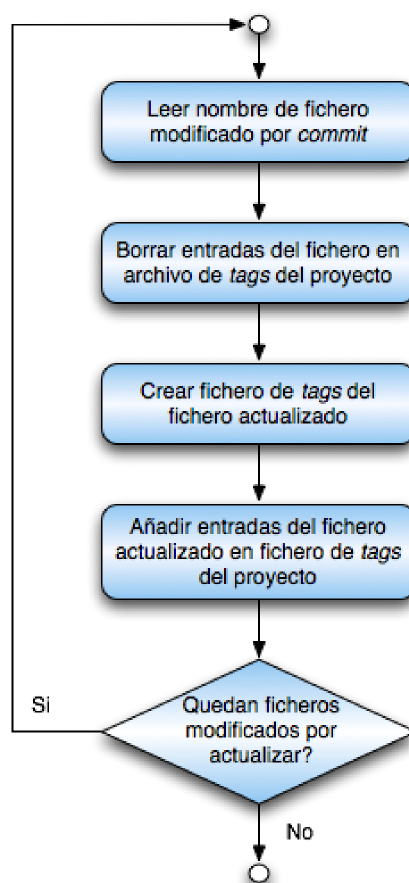

Figura 4.8: Actualización de ficheros de *tags* [Fuente: elaboración propia]

Para ello, recorremos los resultados obtenidos y vamos comparando para encontrar las posibles relaciones una a una como mostramos a continuación (Figura 4.9):

Así vamos recorriendo una a una las líneas que hemos generado en el paso anterior y obteniendo la información relevante que necesitamos para crear las relaciones. De cada línea, queremos buscar coincidencias con otros desarrolladores, por lo que entramos en otro bucle que las busca y

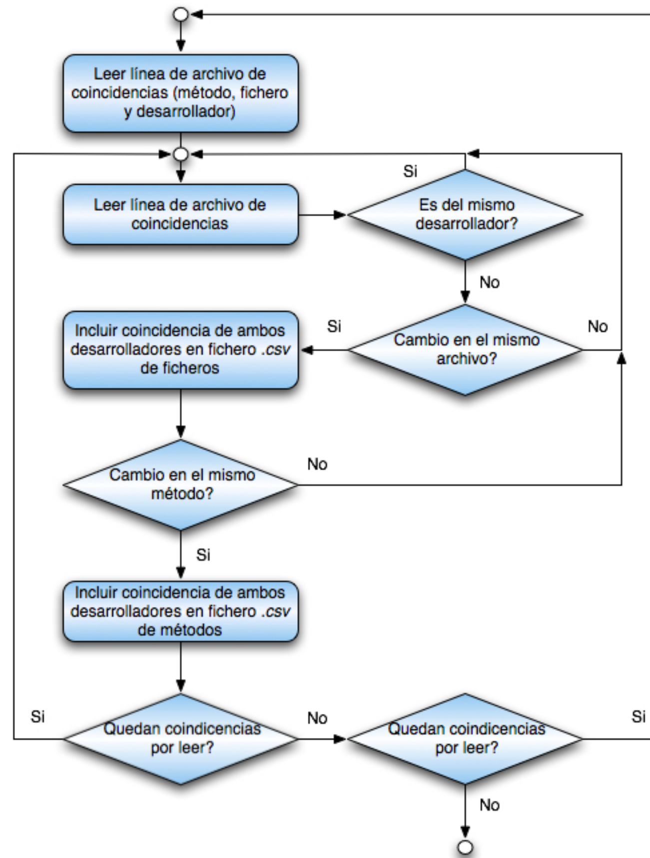

Figura 4.9: Impresión de fichero de asociaciones [Fuente: elaboración propia]

las imprime directamente en el fichero de salida de la siguente forma:

La salida final son dos ficheros en formato *.csv* con las relaciones entre los desarrolladores. Así pueden ser importados por cualquier aplicación que admita este formato para realizar análisis de redes sociales.

Otro dato relevante a conocer a la hora de analizar un proyecto es el número total de desarrolladores que han contribuido a él. Es posible que aquellos

desarrolladores que no han contribuido con frecuencia no tengan una relación notable con otros desarrolladores y por ello no sean tenidos en cuenta en el estudio de la comunidad ni representados en el grafo. Para obtener el número total de desarrolladores, podemos ejecutar un simple comando de Git, y contar el número de ocurrencias existentes que obtenemos como salida:

```
git shortlog -sn | wc -l
```

4.3. Precisión de los resultados

Para dar validez a nuestros resultados, debemos comprobar que, de hecho, estamos aplicando de forma adecuada nuestros algoritmos así como las herramientas usadas. Más concretamente, debemos conocer si los cambios producidos en el código corresponden realmente a métodos en el código fuente de un proyecto.

Para ello debemos analizar dos puntos clave cuando relacionamos cambios con métodos: cuán preciso es *Ctags* al encontrar métodos en un fichero y asignarlo a etiquetas, y cuán preciso es nuestro proyecto al asignar líneas de código de cambios a tales etiquetas.

Una forma tradicional de analizar la precisión de los resultados es utilizando las siguientes medidas [34]:

1. Precisión

   Se define como el ratio de aciertos y fallos a la hora de reconocer un patrón.

$$Precisión = \frac{\{E\} \cap \{N\}}{\{N\}}$$

Siendo E el número de aciertos al reconocer el patrón y N el número de elementos reconocidos de acuerdo al patrón.

2. Exhaustividad

   Se define como el número de elementos encontrados entre el número de elementos existentes cuando se hace una búsqueda a partir de un patrón.

$$Exhaustividad = \frac{\{N\} \cap \{T\}}{\{T\}}$$

   Siendo N de nuevo el número de elementos reconocidos y T el número total de elementos existentes.

3. Valor-F

   A partir de los dos valores anteriores, el Valor-F determina un valor único ponderado que mide la precisión obtenida en un test. En nuestro caso, le damos la misma importancia tanto a la precisión como a la exhaustividad, por lo que el Valor-F se calcula de la siguiente manera:

$$Valor - F = 2 \cdot \frac{\{Precisión\} \cdot \{Exhaustividad\}}{\{Precisión\} + \{Exhaustividad\}}$$

En los tres casos, los parámetros oscilan entre los valores 0 y 1, siendo el 1 el valor máximo posible, que indicaría la exactitud absoluta del resultado, y 0 el mínimo, que significaría una precisión nula.

En nuestro caso concreto, la precisión de *Ctags* es la relación entre aciertos y fallos al encontrar un método en un determinado fichero, y la exhaustividad es el número de métodos encontrados de los totales en un fichero. Para nuestra solución, la precisión en cambio es el ratio de aciertos y fallos realizados entre el número de línea del cambio y el método al que se le asigna tal cambio por parte del *script*. En cuanto a la exhaustividad, es exactamente la misma que para

Ctags, ya que usamos la salida de éste para determinar el número de métodos existentes en un fichero.

De una muestra total de 481 métodos en tres lenguajes de programación diferentes (*PHP*, *Javascript* y *Java*), en los que se han revisado manualmente si nuestra solución ha acertado o fallado al asignar cambios a métodos en el código, hemos obtenido los siguientes datos para *Ctags*:

$$E = 481$$

$$N = 481$$

$$T = 481$$

$$Precisión = \frac{\{481\} \cap \{481\}}{\{481\}} = 1$$

$$Exhaustividad = \frac{\{481\} \cap \{481\}}{\{481\}} = 1$$

$$Valor - F = 2 \cdot \frac{\{1\} \cdot \{1\}}{\{1\} + \{1\}} = 1$$

Para el *script* implementado, los datos son los siguientes:

$$E = 472$$

$$N = 481$$

$$T = 481$$

$$Precisión = \frac{\{472\} \cap \{481\}}{\{481\}} = 0,981$$

$$Exhaustividad = \frac{\{481\} \cap \{481\}}{\{481\}} = 1$$

$$Valor - F = 2 \cdot \frac{\{0'981\} \cdot \{1\}}{\{0'981\} + \{1\}} = 0,99$$

Para nuestro estudio, un Valor-F de 0,99 resulta suficientemente fiable para identificar relaciones entre desarrolladores.

Capítulo 5

Casos de estudio

5.1. Moodle

Moodle es un Sistema de Administración de Cursos ampliamente extendido en todo el mundo. Su primera versión apareció en 2002, por lo que lleva más de 10 años en marcha; su creador, Martin Dougiamas, estudió en una escuela primaria a distancia, lo cual influyó de forma significativa en la creación de este proyecto. El proyecto está licenciado bajo GNU GPL y ha sido traducido en más de 90 idiomas. Los números de Moodle hablan por sí sólos del impacto de esta plataforma: a fecha de enero de 2014, Moodle es usado en 223 países, aloja 7 millones de cursos y tiene más de 65 millones de usuarios. Estados Unidos y España son los dos países con más sitios registrados. Su última versión es la 2.6, aunque la más extendida es la 1.9 [35].

Figura 5.1: Logo de Moodle [35]

El proyecto Moodle da la oportunidad al profesorado de organizar un curso virtual y dispone de todas las herramientas necesaria para la comunicación profesor-alumno. Es utilizado en cursos a distancia, donde por lo general toda la información se almacena en la plataforma, o como apoyo a cursos presenciales. Implementa una gran serie de funcionalidades destinadas a la enseñanza como foros de debate, cuestionarios, alojamiento para material de consulta o tareas. Además es posible obtener informes de la actividad y rendimiento del alumno para una sencilla calificación por parte del profesorado. En definitiva, una plataforma completa para la enseñanza *online*.

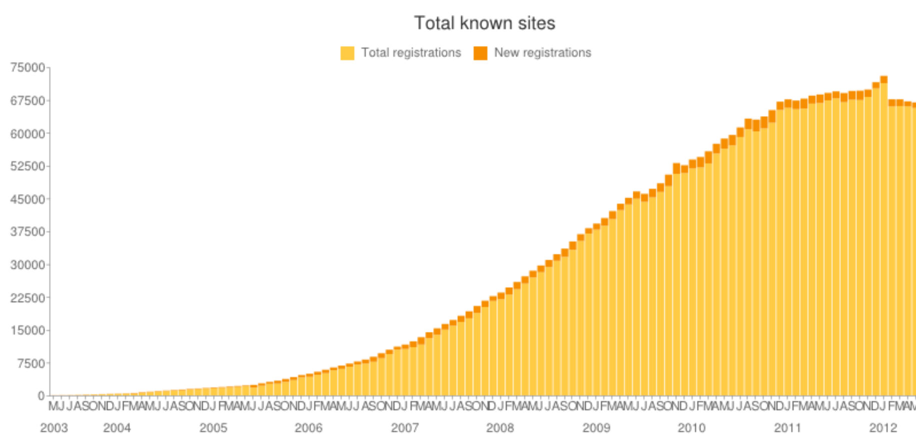

Figura 5.2: Moodle: sitios conocidos [36]

Moodle está desarrollado como una aplicación web para ser ejecutada en cualquier sistema operativo que soporte PHP, e incluye una base de datos SQL. Por lo tanto, todo lo que necesitan los alumnos para el uso de Moodle es una conexión a Internet y un navegador web.

La comunidad del proyecto está formada por más de 1 millón de personas. No obstante, no todas ellas son desarrolladores, ni tampoco todos los desarrolladores tienen permiso para cambiar el código fuente del repositorio. Hay diferentes grupos dentro de la comunidad, los cuales desarrollan roles claramente diferenciados y tienen unos permisos definidos:

- Users
  Pueden crear nuevas incidencias, comentarlos, votar por ellos, documentarlos, y demás tareas para facilitar la labor del desarrollador en el momento de idear e implementar una solución.
- Testers
  Son capaces de probar los nuevos cambios y dar el visto bueno si la funcionalidad se ha desarrollado correctamente.
- Developers
  Pueden asignarse incidencias a sí mismos para trabajar en ellas. Pueden pedir *peer reviews* de la solución implementada a un desarrollador más experimentado, y sólo después del visto bueno de éste, los cambios pueden ser aplicados. El cambio, o *Commit*, lo hace un desarrollador más experimentado.
- Moodle Security
  Desarrolladores de confianza que trabajan en temas de seguridad, los cuales no deben ser vistos por el resto de usuarios.
- Integration requesters

Desarrolladores de más calidad del proyecto y desarrolladores líderes. Disfrutan del nivel más alto de permisos para modificar código.

Como la mayoría de proyectos de Software Libre, Moodle tiene un flujo de trabajo de integración. Este define los pasos a dar desde el momento en el que se encuentra un problema en el software o se requiere la implementación de una nueva funcionalidad. Esto permite llevar un orden en la manera de trabajar dentro del proyecto y mantener un nivel de calidad alto del producto.

El proceso de abrir una incidencia en el sistema de seguimiento de errores del proyecto comienza con la localización de un problema o la identificación de una nueva necesidad. A partir de ese momento, el desarrollador asignado se encarga de resolver la incidencia. Para pasar a la fase de testeo, es necesario pasar dos revisiones: una hecha por un desarrollador común y otra por un desarrollador líder. Durante la fase de testeo, la incidencia no continúa en el caso de que se encuentre un fallo o una falta de funcionalidad. Una vez que se ha comprobado su correcto funcionamiento, la incidencia puede ser cerrada. Para mantener un orden, observamos que las distintas tareas son realizadas en diferentes días de la semana, y que, como ya hemos explicado, cada grupo del proyecto tiene una o varias tareas distintas dentro del proceso.

A partir de los datos almacenados en Git, el software de control de versiones utilizado en el proyecto, podemos obtener un análisis detallado de la comunidad de desarrolladores que colaboran en Moodle. Para ello, como ya hemos explicado, tomaremos dos enfoques; en el primero de ellos, consideramos que una relación entre desarrolladores existe en el momento que dos de ellos han realizado un *Commit* en el mismo fichero del proyecto. En el siguiente enfoque, vamos mas allá y refinamos el concepto de relación entre desarrolladores considerando que han trabajado juntos en el proyecto sólo cuando ambos han modificado el mismo

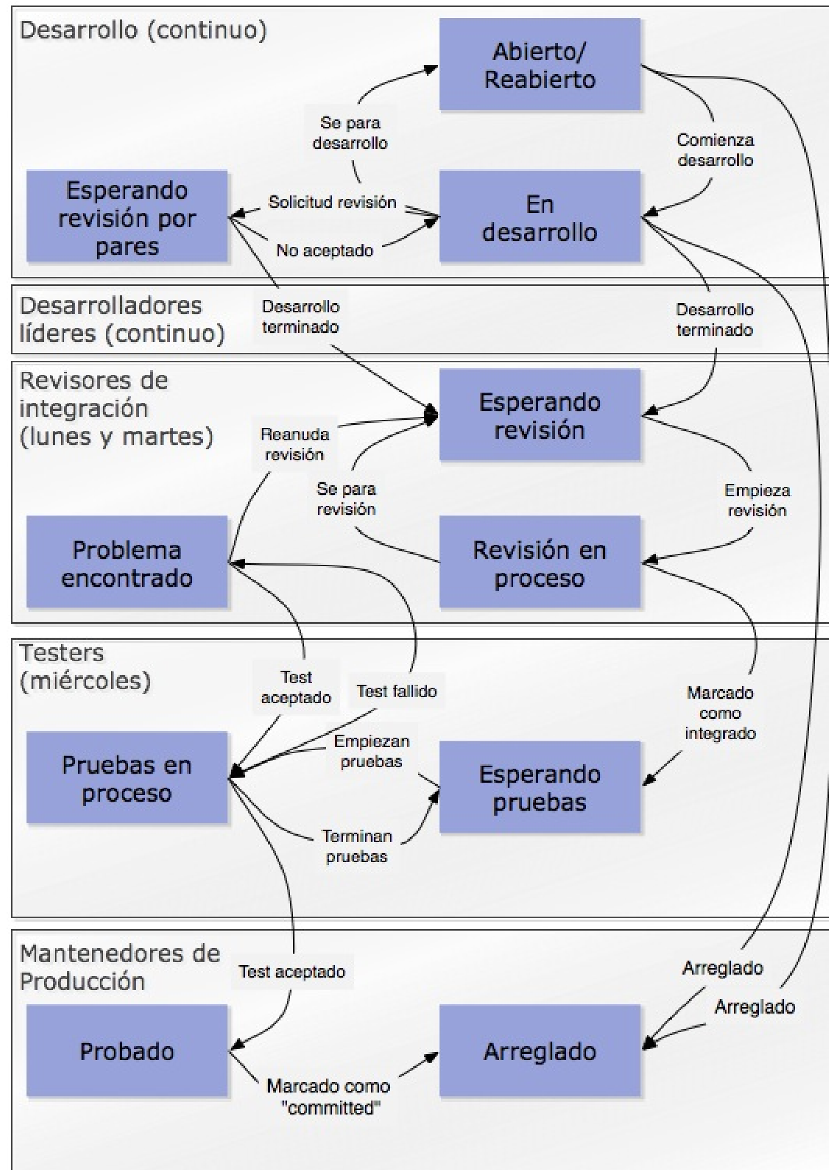

Figura 5.3: Proceso de desarrollo de Moodle [Fuente: elaboración propia]

método o función de un fichero.

En primer lugar, vamos a ver el gráfico de relaciones de la comunidad tomando el primer método, en el que se recogen las relaciones de los desarrolladores entre los meses de abril y junio de 2011, representado en la Figura 5.4.

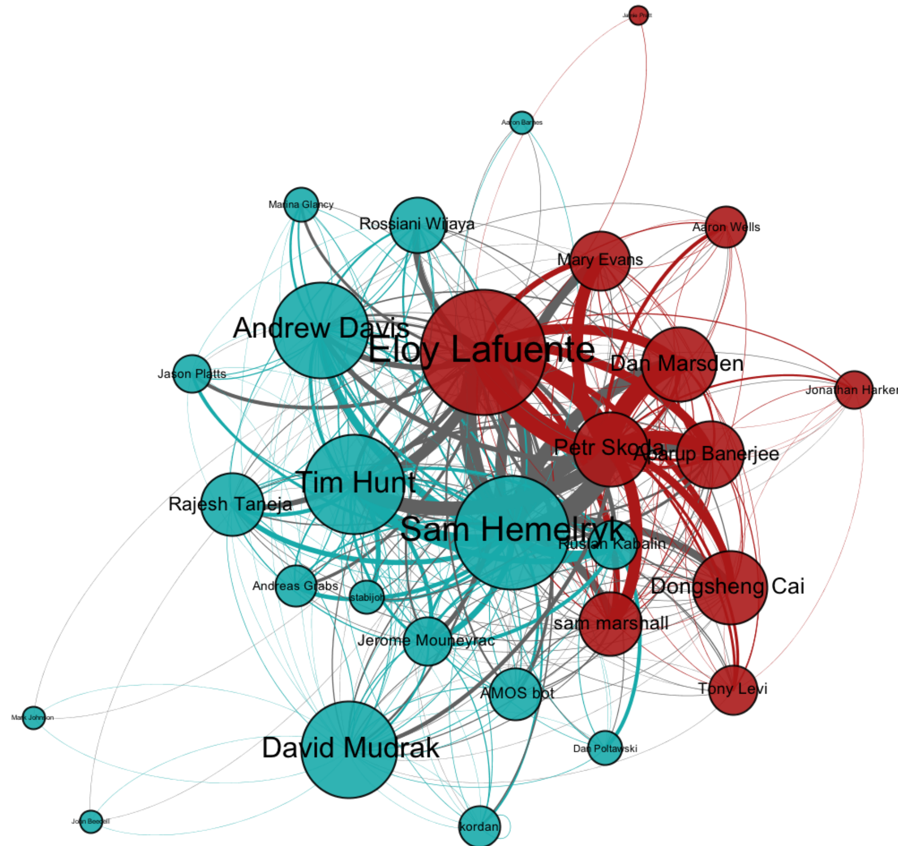

Figura 5.4: Grafo de Ficheros del proyecto Moodle [Fuente: elaboración propia]

El tamaño de los vértices se corresponde con su *betweenness*, por lo que a simple vista podemos ver aquellos desarrolladores más importantes del proyecto. Cabe destacar que el desarrollador con más *commits* puede tener un bajo grado de *betweenness* si otros desarrolladores no han colaborado en la misma

parte del proyecto. La distancia entre los vértices es inversamente proporcional al número de enlaces con otros vértices, por lo que en el centro del grafo tenemos a los desarrolladores con más colaboraciones, mientras que en el exterior se encuentran aquellos con menos relación con otros desarrolladores. También distinguimos desarrolladores de distintos colores; esto se debe a que se ha aplicado el nivel de modularidad del grafo para pintarlos. Los desarrolladores que comparten el mismo color son un grupo dentro de la comunidad. No obstante, en este caso vemos que no se puede diferenciar bien a ambos grupos si tenemos en cuenta la distancia entre ellos, por lo que es muy probable que la modularidad no sea muy alta. De semejante manera, los enlaces tienen el color del grupo que conectan mientras que los grises son enlaces entre desarrolladores de grupos distintos.

La naturaleza del proyecto también influye en la apariencia del grafo. Por ejemplo, en aquellos proyectos en los que se usan varios lenguajes de programación, es probable que la modularidad sea alta y encontremos grupos más diferenciados debido a la especialización de desarrolladores en distintos lenguajes. En cuanto a la organización del proyecto, si analizamos uno en el que sólamente un grupo de desarrolladores líderes tienen permiso para hacer *commit*, el grafo tiende a tener pocos nodos y muy relacionados entre si.

Si usamos el enfoque de relaciones por colaboraciones en métodos en vez de en ficheros, obtenemos el grafo de la Figura 5.5. En primer lugar, cabe destacar la diferencia en el número de nodos y enlaces de ambos grafos. Mientras que en el primer grafo contábamos con 29 desarrolladores, la comunidad del segundo grafo cuenta con 23, resultando en una disminución del 20 %. Las nuevas condiciones para la inclusión de un desarrollador en la comunidad son más estrictas, La diferencia a primera vista es que los nodos con más grado de *betweenness* destacan más; desde este enfoque, la exigencia para que haya un enlace entre dos vértices es mayor; no sólo ambos desarrolladores han de haber trabajado en el

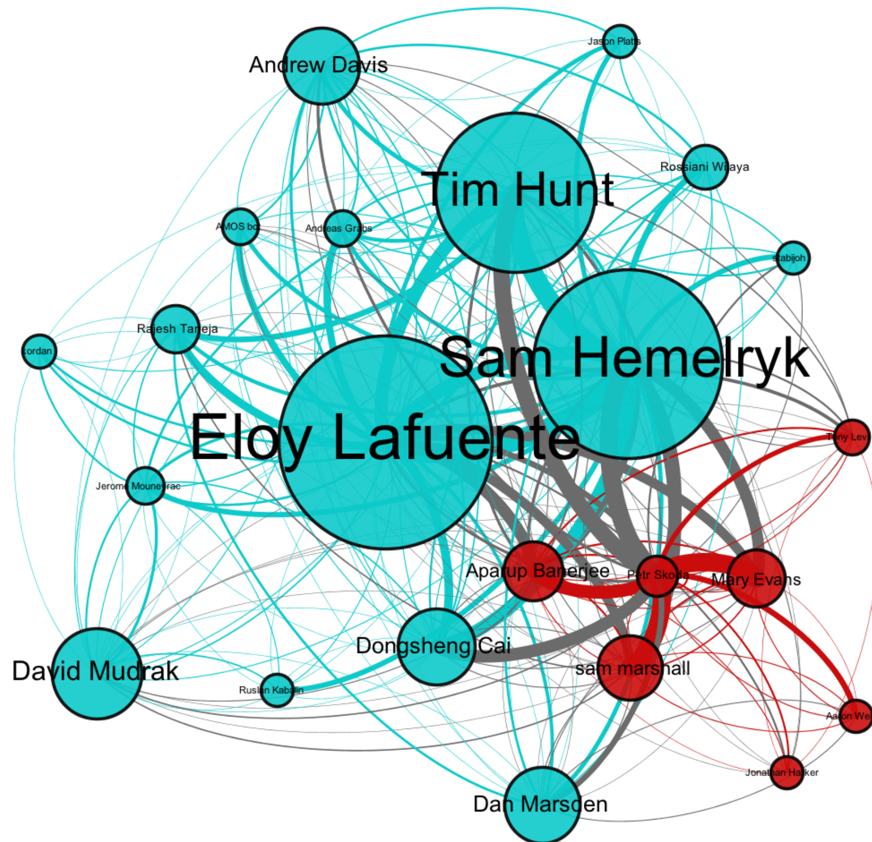

Figura 5.5: Grafo de Métodos del proyecto Moodle [Fuente: elaboración propia]

mismo fichero, sino que además han de haberlo hecho en el mismo método. Por eso aquellos desarrolladores que más han colaborado en el proyecto tienen más probabilidades de tener enlaces con los demás desarrolladores. También vemos cómo se eliminan del grafo los desarrolladores que apenas tenían algún enlace al aplicar el primer enfoque. Por último, observamos cómo el grupo de desarrolladores de color rojo se ha reducido, lo cual nos sugiere que la modularidad no está fuertemente definida, y haciendo un análisis más detallado muchas de las relaciones entre desarrolladores se desvanecen a la vez que se crean otras nuevas,

Parámetro	Ficheros	Métodos
Nodos	29	23
Grado	318	238
Grado Medio	10,966	10,348
Diámetro	2	3
Densidad	0,394	0,502
Modularidad	0,081	0,101
Coeficiente Medio de Clustering	0,829	0,794
Longitud Media de Camino	1,611	1,522

Cuadro 5.1: Parámetros de ambos grafos, proyecto Moodle [Fuente: elaboración propia]

provocando que los grupos también varíen.

A partir de los datos de la Figura 5.1 podemos hacer un análisis de las comunidades usando los parámetros ya descritos. Las conclusiones que podemos obtener de los datos calculados para ambos grafos nos dan una visión objetiva de lo que está ocurriendo en la comunidad. Como ya hemos explicado, se aprecia una disminución del número de nodos en el estudio de la comunidad usando el segundo método, al igual que del orden del grafo. Desde este punto de vista, en el caso del proyecto Moodle, el conjunto de la comunidad tiene un *betweenness* más homogéneo que usando el primer método, mientras que aquellos desarrolladores que menos han aportado a la comunidad se salen del grafo. No obstante vemos un grado medio muy parecido, lo que puede indicar que ambos métodos toman una representación parecida de la comunidad.

Estudiando el parámetro de densidad, podemos reconocer que, a pesar de que el orden del grafo es menor, el nivel de colaboración entre los desarrolladores

Desarrollador	Betweenness por Ficheros	Betwenness por Métodos
Sam Hemelryk	119,6	26,4
Eloy Fuente	244,6	26,4
Tim Hunt	53	23,8
Petr Skoda	14	12,1
David Mudrak	6,8	8,8

Cuadro 5.2: Desarrolladores con mayor Betweenness, proyecto Moodle [Fuente: elaboración propia]

crece en algo más de una décima, debido principalmente al menor número de desarrolladores envueltos en la comunidad. Esto también se ve representado en un descenso de la longitud media de camino, que pierde una décima, revelándonos una relación entre desarrolladores más cercana.

5.2. OpenStack

OpenStack es un proyecto de colaboración en el que están envueltos desarrolladores y profesionales con el objetivo de crear una plataforma de código abierto de computación en la nube. El objetivo del proyecto es crear una solución para todo tipo de nubes (tanto públicas como privadas) con una implementación sencilla, permitiendo una alta escalabilidad y un gran número de características. En realidad, está compuesto por un conjunto de componentes interrelacionados que ofrecen diferentes servicios para crear una infraestructura que ofrece todo tipo de servicios en la nube [37].

Fundado en 2010 por Rackspace Hosting y NASA, forma una comunidad en la que participan más de 200 empresas de primer nivel que colaboran en un proyecto que usan corporaciones, proveedores de servicio, investigadores y

Figura 5.6: Logo de OpenStack [37]

centros de datos entre otros. El código está protegido bajo una licencia Apache 2.0, que permite a cualquiera ejecutar, modificar o estudiar el código formando así un modelo abierto de desarrollo enfocado a dar servicio a proveedores de computación en la nube.

La comunidad de OpenStack desarrolla el proyecto siguiendo una serie de principios [38]:

1. Proceso de diseño abierto

   Cada seis meses la comunidad mantiene una cumbre en la que se habla de requerimientos y especificaciones para la siguiente versión. Las cumbres son abiertas al público.

2. Comunidad abierta

   OpenStack está dedicado a crear una comunidad activa y vibrante de desarrolladores y colaboradores. La mayoría de las decisiones son tomadas

a partir de un modelo de consenso perezoso[1].

3. Modelo de desarrollo abierto

   Todo el código del proyecto OpenStack está disponible de forma gratuita bajo una licencia Apache 2.0.

El proyecto OpenStack está organizado en torno a tres grandes conceptos y servicios compartidos como podemos ver en la figura 5.7.

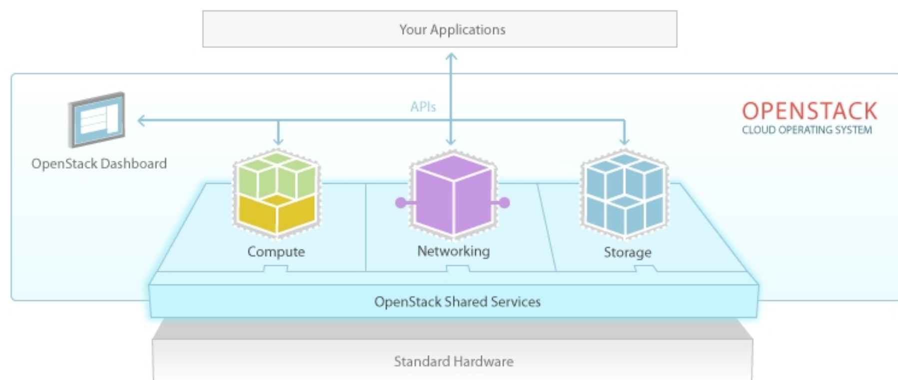

Figura 5.7: Modelo OpenStack [37]

1. OpenStack Computación

   Provisión y manejo de grandes redes de máquinas virtuales. El módulo encargado de ello dentro de OpenStack se llama Nova.

2. OpenStack Almacenamiento

[1] El consenso perezoso consiste en una fórmula para alcanzar acuerdos en la que el silencio se considera el consentimiento de los participantes. De esta forma se ahorra tiempo en discusión, estando los participantes de acuerdo por defecto.

Almacenamiento de objetos (módulo Swift) y almacenamiento de bloques (módulo Cinder) para el uso de servidores y aplicaciones.

3. OpenStack Redes

   Manejo, gestión y dirección de todo tipo de redes. El nombre de este módulo es Neutron.

OpenStack contiene multitud de servicios compartidos que expanden los tres pilares anteriormente descritos, haciendo más fácil la implementación e interoperación de la nube. Estos servicios incluyen gestión de identidades, de imágenes y una interfaz web que provee de una experiencia unificada para que los usuarios interactúen con los diferentes recursos de la nube.

La organización del proyecto es llevada a cabo por la Fundación OpenStack. Ésta promueve el desarrollo, la distribución y la adopción de OpenStack. Provee al completo ecosistema de los recursos necesario para hacer crecer el proyecto. Por supuesto, el acceso a esta fundación es libre y accesible para cualquiera. La fundación es dirigida de la siguiente forma [39]:

- Comité Técnico
  Se compone de 13 miembros, el cual es elegido por los contribuidores activos en la parte ténica del proyecto. Se encarga de determinar los problemas clave de los diferentes módulos.
- Junta Directiva
  Provee de supervisión estratégica y financiera a la fundación.
- Comité de Usuarios
  Con el crecimiento de despliegues de OpenStack en diferentes entornos y la suma de más asociados al proyecto, se vuelve muy importante que

las comunidades que construyen servicios alrededor de OpenStack guíen e influencien el desarrollo del producto. La misión del Comité de Usuarios es:

- Consolidar requerimientos de usuario y presentarlos al Comité Técnico.
- Guiar los equipos de desarrollo que necesitan opiniones de usuarios.
- Realizar un seguimiento de las experiencias de despliegues y uso del producto.
- Trabajar con los grupos de todo el mundo para mantener viva la comunidad.

Para realizar el análisis de la comunidad de OpenStack, hemos tomado datos de contribuciones al proyecto en el módulo Swift en el año 2013. Este módulo implementa un sistema escalable y redundante de almacenamiento. Ficheros y objetos son escritos en múltiples discos duros en servidores de un centro de datos, siendo OpenStack responsable de la replicación de los datos y de su integridad. La escalabilidad se produce de forma horizontal al añadir nuevos servidores. En el momento en el que un disco duro falla, su contenido es duplicado a otros servidores activos. El módulo Swift empezó con un equipo de nueve desarrolladores.

De un total de 146 desarrolladores del módulo, usando el método de asociación por colaboración en el mismo fichero, se han obtenido relaciones entre 89 de ellos, como podemos ver en la Figura 5.8. En cuanto al número de relaciones encontradas, el total han sido 1502. Hemos de mencionar que, en ambos grafos, hemos omitido un *bot* encargado de realizar *commits* de forma automática.

Por otro lado, cuando hemos aplicado el método de colaboraciones entre desarrolladores por métodos, hemos obtenido un total de 974 asociaciones entre 80

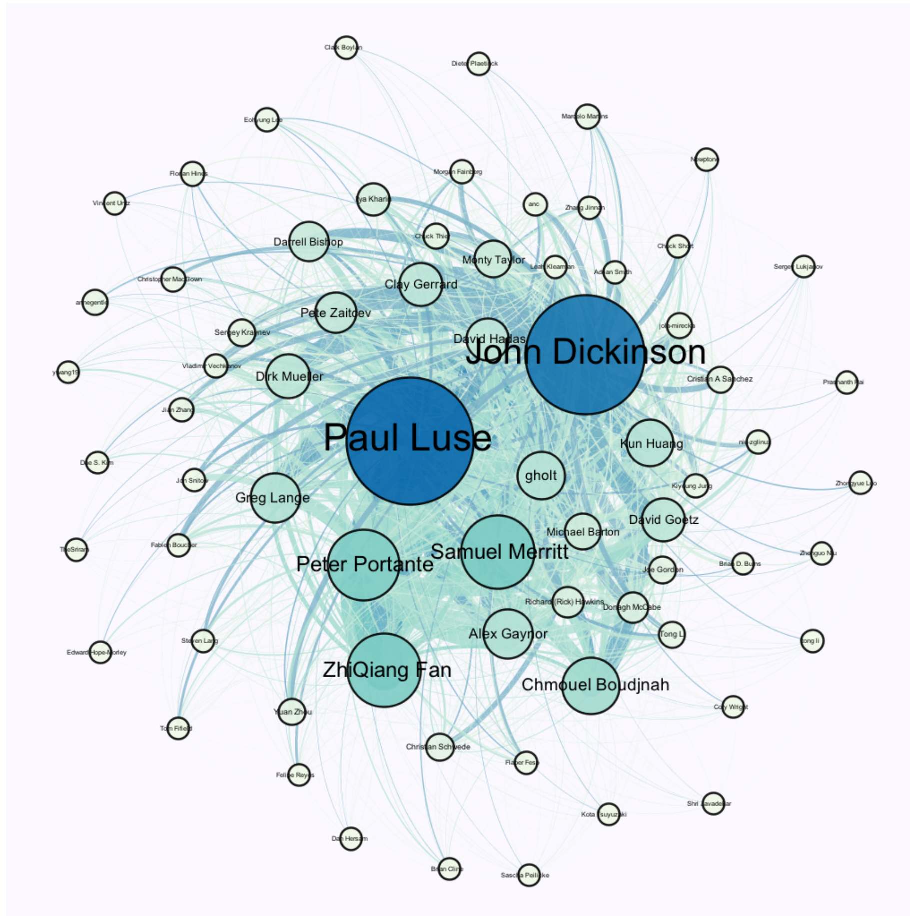

Figura 5.8: Grafo de Ficheros del módulo Swift de OpenStack [Fuente: elaboración propia]

desarrolladores, como se muestra en la Figura 5.9.

La primera diferencia es clara entre ambos métodos: se producen muchas más relaciones con el primero, que toma el fichero como punto de colaboración entre desarrolladores. Esto es obvio si tenemos en cuenta que para que haya una coincidencia en un método primero tiene que haberla en un fichero, siendo el segundo método un subconjunto del primero. Es oportuno mencionar que, de

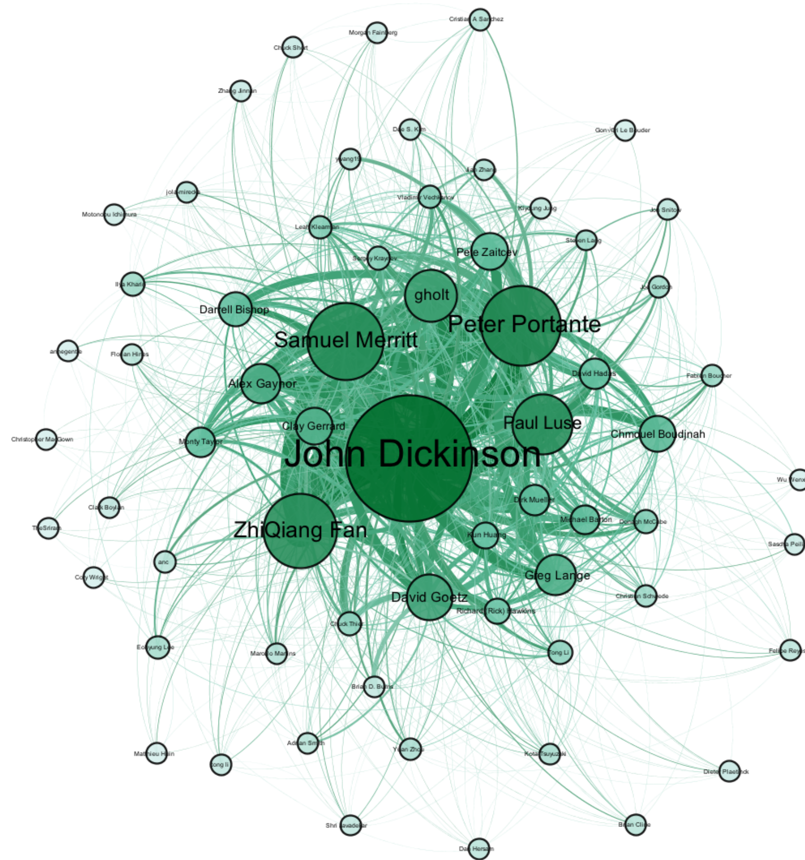

Figura 5.9: Grafo de Métodos del módulo Swift de OpenStack [Fuente: elaboración propia]

los 146 desarrolladores que ha tenido el proyecto a lo largo de su existencia, en ninguno de los casos se supera el 60 % cuando son representados en el grafo; esto puede ser debido a la eventual colaboración de un segmento de desarrolladores que no han producido suficientes cambios en el código como para que su aportación sea notable.

En cuanto a la representación gráfica, la primera característica que salta a la vista es la baja modularidad en las colaboraciones; en ambos casos vemos una red en la que los nodos principales se colocan en el centro debido a su alto *betweenness*, y el resto de colaboradores están esparcidos alrededor de éstos. Se puede observar un cinturón de individuos esparcidos del centro de la red, que representa a aquellos colaboradores que menos han contribuido. El núcleo central de la red es, sin embargo, más abultado en la representación del segundo método a pesar de que el número de desarrolladores representados es similar en ambos; esto provoca que el cinturón exterior esté más poblado.

Aún siendo ambos grafos suficientemente parecidos, hemos de señalar que la calidad de las asociaciones sufren un cambio importante al acotar el espacio en el que se producen las relaciones entre desarrolladores. Reduciendo el ámbito de la relación, las posibilidades de que dos sujetos se conozcan incrementan, y esto se aplica a cualquier comunidad; en nuestro caso, la reducción del ámbito se produce de ficheros a métodos y los sujetos son los desarrolladores de la comunidad de Software Libre estudiada.

Para continuar el análisis, nos fijamos en los parámetros resultantes del estudio de la red que podemos observar en el cuadro 5.3.

Al observar el grado medio de varios métodos, podemos distinguir una diferencia significativa. De acuerdo con el segundo método, las relaciones entre los desarrolladores no son tan intensas. Esto se debe al decrecimiento del número de enlaces mientras que el número de desarrolladores se mantiene. En cuanto al diámetro, es un punto superior en el primer método, lo cual resulta sorprende si tenemos en cuenta que el grado medio es más alto con este método. La representación de la red con el segundo método, por lo tanto, es más compacta. La modularidad es nula usando ambos métodos. La única conclusión que po-

Parámetro	Ficheros	Métodos
Nodos	89	80
Grado	1502	974
Grado Medio	33,75	24,35
Diámetro	3	2
Densidad	0,384	0,308
Modularidad	0	0
Coeficiente Medio de Clustering	0,823	0,834
Longitud Media de Camino	1,618	1,692

Cuadro 5.3: Parámetros de ambos grafos, proyecto OpenStack [Fuente: elaboración propia]

demos sacar de este resultado es que ambos métodos representan de la misma manera la estructura centralizada en este caso del proyecto OpenStack. Como consecuencia del menor número de enlaces en el segundo método, la densidad del grafo también es menor. Esto significa que tenemos una comunidad menos interconectada.

Un resultado interesante que hemos obtenido es el coeficiente medio de *Clustering*, el cual es ligeramente mayor usando el segundo método. Esto significa que, a pesar de obtener una red con menos enlaces, los individuos parecen comunicarse con aquellos más relevantes de forma más frecuente. La longitud media de camino es también mayor en el segundo método. Este resultado puede deducirse del grafo de la comunidad al ver que los individuos aparecen más dispersos. Por lo tanto, es más grande el costo de relacionarse con otro desarrollador dentro del proyecto según la representación del segundo método.

Desarrollador	Betweenness por Ficheros	Betwenness por Métodos
John Dickinson	271	351’8
Peter Portante	137	196’6
Samuel Merrit	144,2	186’2
Zhiqiang Fan	145	176’2
Paul Luse	293	130’8

Cuadro 5.4: Desarrolladores con mayor Betweenness, proyecto Moodle [Fuente: elaboración propia]

Para observar el *betweenness* de los desarrolladores más influyentes, vamos a analizar el cuadro 5.4. En él, vemos como usando el primero método obtenemos dos desarrolladores con una gran influencia por delante de los demas; John Dickinson y Paul Luse. En cambio, usando el segundo, hay dos desarrolladores muy influentes y un segundo grupo que tiene una influencia importante por detrás de estos. Es sorprende ver cómo el desarrollador que se queda fuera de este grupo de cabeza era el segundo más influyente en el primer caso y ahora ha pasado a la cola del segundo grupo. La pérdida de importancia de este desarrollador puede ser debida a que, a pesar de haber contribuido mucho al código como demuestra el primer método, lo haya hecho sin muchas relaciones con otros desarrolladores.

5.3. Webkit

WebKit es el motor sobre el que funcionan importantes navegadores web como Opera, Google Chrome o Safari, el cual fue desarrollado a partir del navegador web del proyecto KDE, Konqueror. Está desarrollado bajo varias licencias como son GNU GPL, GNU LGPL y BSD. Además de navegadores web, WebKit también es usado en otros ámbitos como el sistema operativo para móviles Android,

la plataforma de distribución digital de videojuegos Steam o el cliente de mensajería instantánea Adium. Entre los desarrolladores del proyecto se encuentran empresas como Apple Inc., Qt Sotware, Adobe, Nokia o Google [40].

Figura 5.10: Logo de WebKit [41]

WebKit provee de una serie de herramientas específicas para desarrolladores, entre ellas un analizado sintáctico y motor de renderizado de HTML, y un intérprete de Javascript, además de aquellas herramientas propias en proyectos de Software Libre [42].

Como podemos ver en la Figura 5.11, la cuota de mercado de aquellos navegadores que usan WebKit es, aproximadamente, de un 50 %.

Al ser un proyecto usado por varias de las mayores empresas de software del mundo, éstas disponen recursos a disposición de WebKit para mejorar el proyecto. Además, en el caso de este proyecto, los nombres de usuarios registrados en Git contienen también el dominio de correo que éstos usan, el cual coincide con el nombre de su empresa, por lo que también somos capaces de ver qué empresas contribuyen más al proyecto.

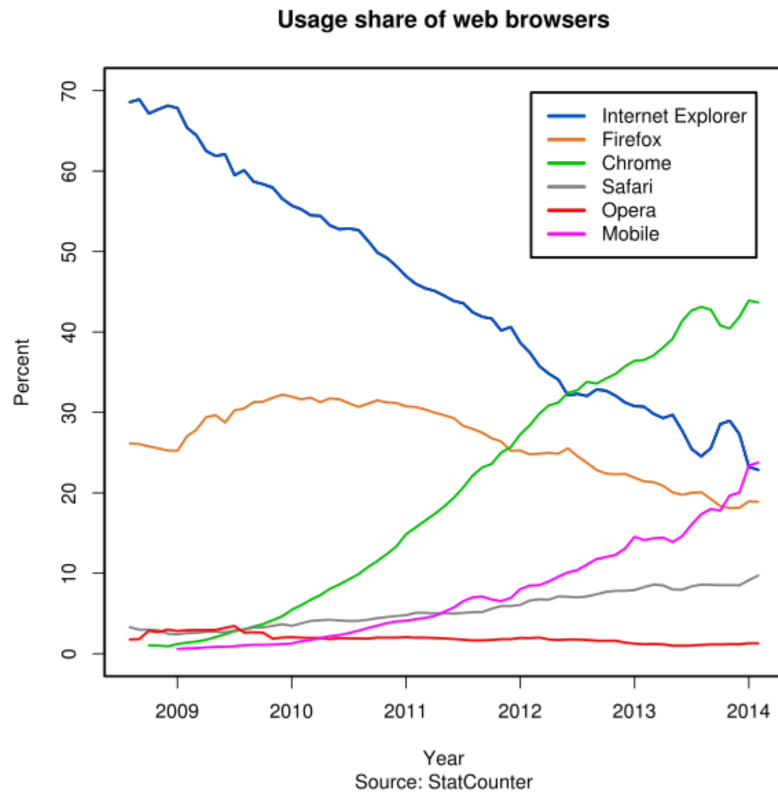

Figura 5.11: Cuota de mercado de navegadores web [43]

En la Figura 5.12 mostramos el grafo con la representación de la comunidad de desarrolladores según el primer método de análisis durante los primeros seis meses del año 2013. Como podemos ver, hay una gran homogeneidad en cuanto al *Betweenness* de los desarrolladores, que está representado de acuerdo al tamaño de los nodos del grafo. Al contrario de otros proyectos, ninguno de los desarrolladores destaca por encima de los demás, y ni siquiera hay un grupo de ellos, los que normalmente suelen ser los desarrolladores líderes del proyecto, con más importancia que el resto de la comunidad.

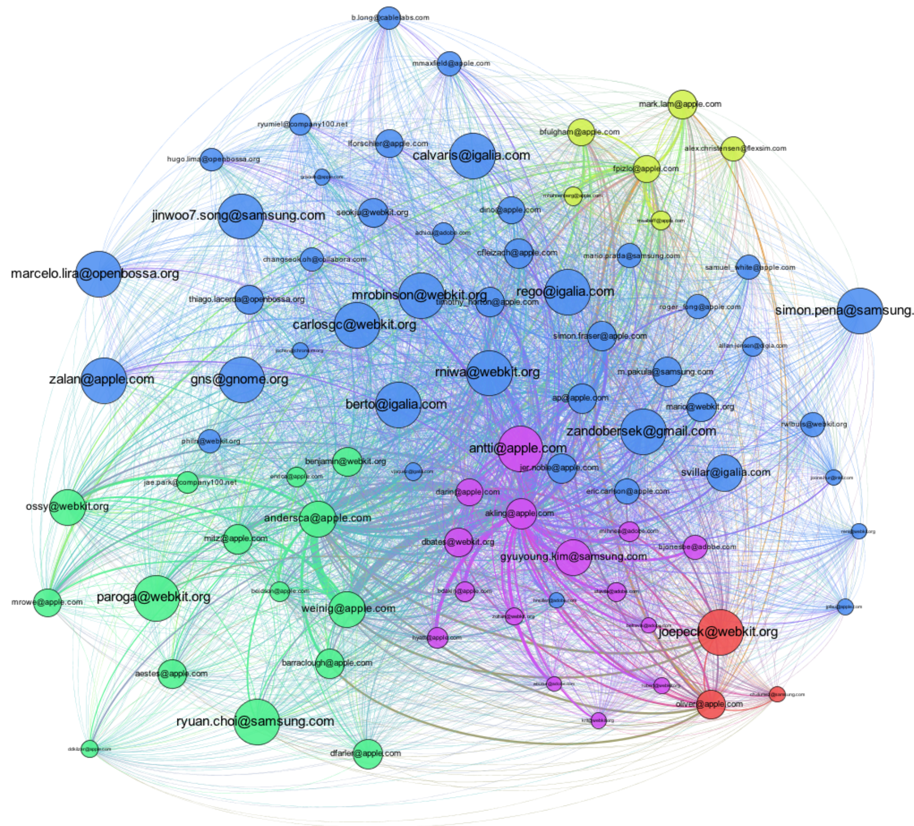

Figura 5.12: Grafo de ficheros del proyecto WebKit [Fuente: elaboración propia]

Por otro lado, podemos ver claramente los grupos que forman la comunidad, los cuales están diferenciados con distintos colores. A partir del grafo podemos deducir que la comunidad de WebKit destaca por su modularidad. Hasta cinco grupos distintos pueden ser diferenciados. No obstante, vemos que éstos no están radicalmente separados del resto de los desarrolladores, sino que también tienen numerosos enlaces con desarrolladores de otros grupos. Esto se puede deber a los distintos grupos de trabajo dentro del proyecto, como *WebKitEFL*, el cual es

un conjunto de librerías para interfaces gráficas de usuario [44], *WebKitGTK+*, que se centra en el renderizado web para *GNOME* [45] o QtWebKit, que se encarga de la portabilidad con la librería *Qt* [46].

En la imagen 5.13 podemos ver el grafo según el criterio de relaciones por colaboraciones en el mismo método.

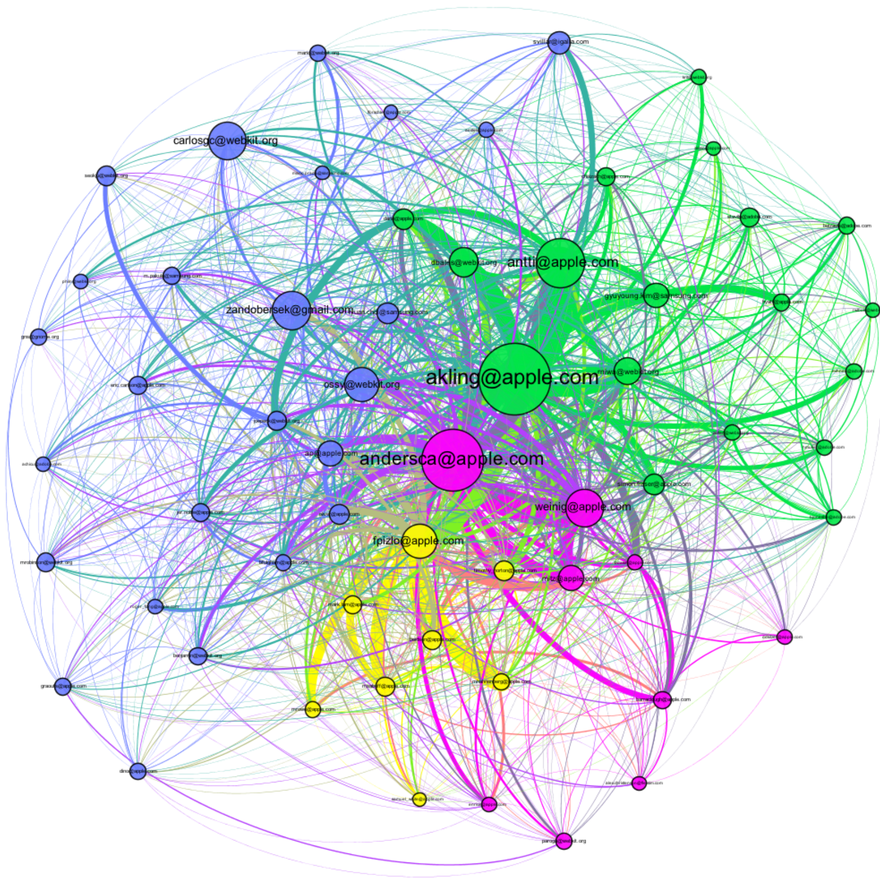

Figura 5.13: Grafo de métodos del proyecto WebKit [Fuente: elaboración propia]

Cuando analizamos el grafo de métodos nos encontramos con una realidad notablemente distinta. Podemos ver en primer lugar que esta vez hay varios desarrolladores que destacan sobre los demás. Con la representación del grafo de métodos destacan desarrolladores que presumiblemente pueden haber contribuido más al código del proyecto. También observamos en el grafo un cinturón de desarrolladores de menos importancia, creando así una comunidad más heterogénea que en la representación anterior. También notamos una ligera diferencia en cuanto a la modularidad dentro de la comunidad, que ahora son cuatro en vez de cinco. Aún así, siguen siendo unos grupos que tienen bastante relación con otros desarrolladores fuera de su ámbito.

No debemos sacar la conclusión de que, debido a que las distancias entre nodos en ambas representaciones son similares, las probabilidades de que se conozcan los desarrolladores son las mismas. Como ya hemos explicado con anterioridad, el segundo grafo representa una comunidad de desarrolladores en la que es más probable que los desarrolladores se conozcan debido a la reducción del ámbito de búsqueda de las relaciones.

Más allá de la representación gráfica, cuando realizamos un análisis cualitativo de la comunidad, también encontramos diferencias significativas como podemos ver en el cuadro 5.5. En primer lugar, hay una ligera disminución en el número de desarrolladores en la comunidad. De todas formas, no es comparable a la reducción del número de enlaces, que es aproximadamente de cuatro a uno. Esto provoca una comunidad menos comunicada, como podemos corroborar con el dato de densidad. Mientras que en el método de asociaciones por ficheros es de 0,85, que significa que prácticamente todos los desarrollores se conocen, en el método de asociaciones por métodos es de 0,25. Es poco realista pensar que casi todos los desarrolladores de una comunidad de Software Libre se conocen, ya que muchos de ellos no llegan siquiera a conocerse personalmente, dificultando así las asociaciones entre ellos. El diámetro es un nivel superior en el método de

Parámetro	Ficheros	Métodos
Nodos	130	114
Grado	7166	1735
Grado Medio	110,2	30,43
Diámetro	2	3
Densidad	0,85	0,26
Modularidad	0,07	0,12
Coeficiente Medio de Clustering	0,921	0,764
Longitud Media de Camino	1,145	1,77

Cuadro 5.5: Parámetros de ambos grafos, proyecto WebKit [Fuente: elaboración propia]

asociaciones por métodos. La diferencia de resultados del diámetro de los grafos de los casos propuestos nos da un primer indicio a pensar que este valor no es significante, ya que difiere sin ningún sentido aparente.

A pesar de haber un grupo menos en la comunidad, la modularidad es mayor en el segundo grafo. Eso significa que los grupos aquí son más compactos, o en otras palabras, que hay más relaciones de los desarrolladores dentro de su grupo que fuera de él si lo comparamos con el primer grago. En cuanto al coeficiente medio de clustering, vemos que es un punto y medio superior usando el primero método; esto se debe a que, como hemos observado, la comunidad del primer grafo resulta ser más homogénea. Por lo tanto las conexiones entre los desarrolladores tienen más importancia. En cambio en el segundo grupo, hay un conjunto de desarrolladores que no han colaborado tanto, por lo que la media desciende. La longitud media de camino mayor en el segundo grafo también se explica con el resultado de la densidad; cuantas menos conexiones, más complicado es alcanzar el resto de nodos del grafo.

Para analizar las características de los desarrolladores, hemos tomado una muestra más extensa de los desarrolladores más importantes del proyecto de WebKit para obtener una perspectiva de la comunidad usando ambos métodos.

Desarrollador	Betweenness por Ficheros	Betwenness por Métodos
aklin@apple.com	15	519
andersca@apple.com	18,6	434
antti@apple.com	24,8	316
zandobersek@gmail.com	24,8	220
weinig@apple.com	18,6	215
carlosgc@webkit.org	24,8	211
fpizlo@apple.com	12,9	185
ossy@webkit.com	18,6	180
dbates@webkit.com	14,3	138
rniwa@webkit.com	24,8	114
ap@apple.com	14,3	106
mitz@apple.com	14,3	104
gyuyoung@samsung.com	18,6	102
rego@igalia.com	24,8	2
berto@igalia.com	24,8	-
svillar@igalia.com	18,6	78
simon.fraser@apple.com	14’3	66
mrobinson@webkit.com	24’8	45
paroga@webkit.com	24’8	23

Cuadro 5.6: Desarrolladores con mayor Betweenness en WebKit [Fuente: elaboración propia]

Lo primero que notamos al mirar el cuadro 5.6 es que la comunidad creada con el segundo método contiene desarrolladores con un alto *Betweenness* a diferencia de los del primer método, como también se puede deducir a partir de los grafos. Un dato destacable es que algunos de los desarrolladores con mayor *Betweenness* del segundo método no están entre los que más tienen del primero (nótese que están ordenados de mayor a menor según el *Betweenness* del segundo método, y que el *Betweenness* máximo del primer método es 24,8). Incluso uno de los desarrolladores que tienen el máximo *Betweenness* encontrado con el primer método ni siquiera aparece en el grado del segundo, como es el caso de Berto de Igalia. Esta diferencia tan importante nos da un indicio de que algunos desarrolladores pueden haber colaborado mucho al proyecto, pero quizás no tiene muchas relaciones en la comunidad.

Al usar el primer método, se han detectado diecisiete desarrolladores con un *Betweenness* máximo de 24,8. De ellos, de acuerdo al dominio del correo electrónico configurando en Git, cinco tienen un dominio de correo electrónico de WebKit, tres de Igalia, tres de Samsung y dos de Apple entre otros. Hay que tener en cuenta que, de los cinco desarrolladores con dominio de correo de WebKit, dos de ellos son de Igalia (carlosgc@webkit.org y mrobinson@webkit.org) y uno de ellos de Apple (rniwa@webkit.org). No estamos teniendo en cuenta tampoco en este caso los *bots* del proyecto. De ahí podríamos deducir que el grupo que más trabaja en el código es el perteneciente a Igalia. Pero si nos fijamos en la columna del segundo método, vemos que los desarrolladores que más importancia tienen son los que pertenecen a Apple. Si le sumamos que hemos dejado fuera de la tabla a siete desarrolladores con un *Betweenness* entre cien y cincuenta por razones de espacio, de los cuales cinco son de Apple, llegamos a la conclusión de que ésta es la empresa que contribuye al proyecto, ya que la mayoría de los desarrolladores con importancia en la comunidad pertenecen a ésta.

Capítulo 6

Conclusiones

6.1. Resultados

La primera y más clara conclusión que podemos sacar es que ambos métodos son más parecidos que diferentes. Debido a que es más difícil encontrar coincidencias en métodos que en ficheros (de hecho el conjunto de coincidencias en métodos es un subconjunto de las coincidencias en ficheros), el grafo es notablemente menor si tomamos una muestra del mismo periodo de tiempo. Aún así, los datos que obtenemos, nos permiten analizar las comunidades sin este problema. Además usando el segundo método la calidad de las relaciones aumenta, por lo que la certeza de que dos desarrolladores se conozcan está mejor representada usando éste en vez del de relaciones por colaboración en ficheros.

Las diferencias parecen encontrarse a medida que el proyecto se hace más grande; debido a la gran cantidad de *commits* de los proyectos, acaba pareciendo como si todos los desarrolladores se conocieran entre ellos. En el proyecto de WebKit hemos visto que la densidad y el grado medio del grafo de métodos es

notablemente menor, añadiendo un plus de realidad a la representación de la comunidad. El método de asociaciones de desarrolladores por colaboración en métodos es simplemente más selectivo a la hora de crear enlaces.

Otra conclusión que podemos extraer es el alto nivel de entropía de las comunidades; de ahí los grafos tan centralizados que obtenemos, en el que todos los desarrolladores parecen conocerse entre sí sin dividirse claramente en diferentes grupos. Posiblemente la naturaleza del Software Libre, que permite total libertad a los desarrolladores para trabajar en el aspecto que prefieran, sea la causa de este alto nivel de anarquía.

Las diferencias también las encontramos cuando observamos los desarrolladores de forma individual en vez de observar la comunidad completa. Algunos de ellos, aplicando el método de asociación por colaboración en funciones, han perdido importancia en la red, mientras otros la han conseguido. Esto también puede reflejar que con tal método se le da mayor importancia a aquellos programadores puros, ya que hay ficheros en muchos proyectos que simplemente no contienen funciones (documentación, ficheros de configuracion, etc.). De esta forma podemos afirmar que con el nuevo método consideramos a aquellos que han participado en el código como sujetos más importantes en la comunidad.

Podemos añadir que el nivel de *Betweenness* que da como resultado usando el segundo método es más heterogéneo que usando el primero, también posiblemente como consecuencia de que es más selectivo con las asociaciones.

En cuanto a la implementación, la característica principal que hemos perseguido es la exactitud de los datos. Como ya hemos analizado y explicado (véase apartado 4.3), el grado de fiabilidad de la implementación es suficientemente alto como para confiar en nuestro análisis. Hemos de destacar también la sencillez

que presenta esta implementación de cara al usuario. No es necesario instalar otras herramientas intermedias para obtener los datos necesarios para realizar un análisis completo; simplemente se necesita el código fuente y el *script* que proporcionamos, que ha de ejecutarse en el mismo directorio.

6.2. Líneas futuras

Respecto a las líneas futuras, se podría pensar que un análisis de las líneas de código modificadas podría sumar más calidad a la búsqueda de relaciones. No obstante, debido a que un método ya representa la unidad mínima de un problema, no sería realmente representativo ya que la comunicación se produce más en la dirección de resolver problemas como un todo. Si sumamos la complejidad que puede suponer encontrar las líneas exactas modificadas, la precisión podría ser notablemente más baja.

Un aspecto que podría ser estudiadio sería el conjunto de todas las relaciones que quedan registradas entre los desarrolladores. Tanto el código en el sistema de control de versiones tanto las listas de correo o los comentarios hechos en los sistemas de seguimiento de errores pueden ofrecer, juntos, una información muy precisa de las comunicaciones llevadas a cabo en un proyecto. También se pueden tomar en cuenta otros factores como pertenencia a la misma empresa o localización geográfica, que pueden ser facilitadores de las relaciones entre desarrolladores.

6.3. Limitaciones

En nuestro proyecto nos centramos en las relaciones entre desarrolladores. Sin embargo, si se que quiere analizar el comportamiento de una comunidad de

Software Libre a través de su repositorio de software, nos estamos olvidando de todos aquellos contribuidores que colaboran de otra forma al proyecto, como por ejemplo traduciendo, documentando, testeando o financiando el proyecto. Para realizar una análsis completo, han de analizarse también otros medios de comunicación como el correo electrónico o el sistema de seguimiento de errores. Incluso de esta forma es extremadamente complicado obtener una representación que se ajuste a la realidad en el cien por cien de los casos.

Una de las desventajas es que los proyectos deben estar alojados en un repositorio Git. Aún que la mayoría de ellos lo están, y el crecimiento de Git resulta muy prometedor de cara al futuro como principal sistema de control de versiones, algunos otros proyectos importantes de Software Libre usan otros sistemas como *CVS*, *Mercurial* o *SVN*. Por ello, sería necesario usar alguna herramienta intermedia que migrara el código a un repositorio Git para poder realizar análisis con nuestra herramienta.

En cuanto a la implementación, el principal problema es el tiempo que tarda en ejecutarse el *script*. El tiempo de ejecución es directamente proporcional a los cambios hechos en el código, por lo que aquellos proyectos más grandes tardan más en ser analizados. Por poner un ejemplo, para proyectos de gran tamaño, realizar un análisis de dos meses puede tardar un día entero. A pesar de los esfuerzos realizados para disminuirlo, el tiempo que se tarda es aún extenso por lo que, en caso de querer una muestra de datos de un gran espacio de tiempo, se recomienda ejecutarlo en una máquina que no tenga que ser apagada con frecuencia.

Capítulo 7

Bibliografía

[1] J. M. González-Barahona, G. Robles, and J. S. Pascual, *Introducción al Software Libre*. UOC, November 2003.

[2] W3Techs, "Usage of Web Servers for Websites." http://w3techs.com/technologies/overview/web_server/all, February 2014.

[3] Z. Johnson, "WordPress Continues to Dominate in Market Share." http://www.bloggingtips.com/2013/10/03/wordpress-continues-dominate-market-share/, October 2013.

[4] J. Dougherty, "Infographic: WordPress owns 66 % of CMS market, powers 15 % of websites." http://leaderswest.com/2013/05/28/infographic-wordpress-has-66-of-the-cms-market/, May 2013.

[5] Top500, "Development Over Time - Operating System Family - Systems Share." http://www.top500.org/statistics/overtime/, November 2013.

[6] L. López-Fernández, G. Robles, and J. M. González-Barahona, "Applying social network analysis to the information in CVS repositories," *1st Inter-*

national Workshop on Mining Software Repositories (MSR), pp. 101–105, 2004.

[7] GSYC, "CVSAnalY: Source code management tool analyzer." `http://tools.libresoft.es/cvsanaly`, July 2011.

[8] G. Madey, V. Freeh, and R. Tynan, "The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory," *Eighth Americas Conference on Information Systems*, 2002.

[9] J. Xu, S. Christley, and G. Madey, "Application of Social Network Analysis to the Study of Open Source Software," *The Economics of Open Source Software Development*, 2006.

[10] Source Forge, "SourceForge - About." `http://sourceforge.net/about`, 2014.

[11] L. López-Fernández, G. Robles, J. M. González-Barahona, and I. Herraiz, "Applying Social Network Analysis Techniques to Community-Driven Libre Software Projects," *International Journal of Information Technology and Web Engineering (IJITWE)*, vol. 1, no. 3, pp. 27–48, 2006.

[12] J. Martínez-Romo, G. Robles, J. M. González-Barahona, and M. Ortuño-Pérez, "Using Social Network Analysis Techniques to Study Collaboration between a FLOSS Community and a Company," *IFIP âĂŞ The International Federation for Information Processing*, vol. 275, pp. 171–186, 2008.

[13] P. Wagstrom, J. Herbsleb, and K. Carley, "A Social Network Approach to Free/Open Source Software Simulation," *Proceedings of the 1st International Conference on Open Source Systems*, 2005.

[14] K. Crowston and J. Howison, "The social structure of Open Source Software development teams," *International Conference on Information Systems*, 2003.

[15] Exuberant Ctags, “What is ctags?.” http://ctags.sourceforge.net/whatis.html, 2009.

[16] S. Chacon, “Getting Started - A Short History of Git.” http://git-scm.com/book/en/Getting-Started-A-Short-History-of-Git, 2009.

[17] Andrés, “Buenas prácticas en Git.” http://inqbation.com/es/buenas-practicas-en-git/, 2014.

[18] Ohlot.net, “Compare Repositories Market Share.” http://www.ohloh.net/repositories/compare, February 2014.

[19] K. Finley, “Github Has Surpassed Sourceforge and Google Code in Popularity.” https://web.archive.org/web/20130825053520/http://readwrite.com/2011/06/02/github-has-passed-sourceforge, June 2011.

[20] D. Goode, “Scaling Mercurial at Facebook.” https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/, January 2014.

[21] M. A. Valero, “Comparacion Git vs SVN.” http://prezi.com/gadlpkmhmjjj/comparacion-git-vs-svn/, June 2013.

[22] A. Blewitt, “Mercurial and Git: a technical comparison.” http://alblue.bandlem.com/2011/03/mercurial-and-git-technical-comparison.html, March 2011.

[23] N. Hamilton, “The A-Z of Programming Languages: BASH/Bourne-Again Shell.” http://www.computerworld.com.au/article/222764/a-z_programming_languages_bash_bourne-again_shell/?pp=2&fp=16&fpid=1, May 2008.

[24] N. Hamilton, “Bash is in beta release.” https://groups.google.com/forum/?hl=en#!msg/gnu.announce/hvhlR1Vn1P0/NYwp-4_0CaUJ, May 2008.

[25] R. Stallman, "El proyecto GNU." http://www.gnu.org/gnu/thegnuproject.html, October 2010.

[26] Perl.org, "Online Perl Documentation." http://www.perl.org/docs.html, 2014.

[27] M. Bastian, "Gephi: An Open Source Software for Exploring and Manipulating Networks," *AAAI Publications, Third International AAAI Conference on Weblogs and Social Media*, 2009.

[28] Gephi.org, "Gephi, an open source graph visualization and manipulation software." https://gephi.org/, 2014.

[29] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," *International AAAI Conference on Weblogs and Social Media*, 2009.

[30] Gephi.org, "Gephi Tutorial Quick Start." http://gephi.org/tutorials/gephi-tutorial-quick_start.pdf, March 2010.

[31] R. Tanase and R. Radu, "HITS Algorithm - Hubs and Authorities on the Internet." http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture4/lecture4.html, 2009.

[32] S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Computer Networks and ISDN Systems*, vol. 30, pp. 107–117, 1998.

[33] G. Robles, S. Koch, and J. M. González-Barahona, "Remote analysis and measurement of libre software systems by means of the CVSAnalY tool," *2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, pp. 51–54, 2004.

[34] D. M. W. Powers, "Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness and Correlation," vol. 2, pp. 37–63, 2007.

[35] Moodle.org, "About Moodle." http://docs.moodle.org/26/en/About_Moodle, 2014.

[36] Moodle.org, "Moodle Statistics." https://moodle.org/stats/, 2014.

[37] OpenStack Foundation, "OpenStack Community." https://www.openstack.org/community/, 2013.

[38] OpenStack Foundation, "OpenStack Welcome Guide." https://www.openstack.org/assets/welcome-guide/OpenStackWelcomeGuide.pdf, 2013.

[39] OpenStack Foundation, "OpenStack Foundation." http://www.openstack.org/foundation/, 2013.

[40] Bitelia, "Opera 15 cambia por dentro y por fuera." http://bitelia.com/2013/07/novedades-de-opera-15, July 2013.

[41] The WebKit Open Source Project, "Welcome to the website for the WebKit Open Source Project!." http://www.webkit.org/, 2014.

[42] M. A. Álvarez, "Qué es WebKit." http://www.desarrolloweb.com/articulos/que-es-webkit.html, July 2008.

[43] StatCounter, "Stat Counter Global Stats." http://gs.statcounter.com/, 2014.

[44] The WebKit Open Source Project, "EFL Port of WebKit." http://trac.webkit.org/wiki/EFLWebKit, 2014.

[45] The WebKit Open Source Project, "Web content rendering for the GNOME Platform." http://webkitgtk.org/, 2014.

[46] The WebKit Open Source Project, "Qt Port of WebKit." https://trac.webkit.org/wiki/QtWebKit, 2014.