Editor Colaborativo

Proyecto Sistemas Informáticos

Facultad de Informática

Universidad Complutense de Madrid

Curso 2010-2011

Fausto Martín López,
Jesús Pedrosa Guerrero y
Luis Sánchez González

Directores: César Andrés Sánchez y Luis Llana Díaz
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Autorización de difusión

Los abajo firmantes, matriculados en SISTEMAS INFORMÁTICOS de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo realizado durante el curso académico 2010-2011 bajo la dirección de los profesores César Andrés Sánchez y Luis Llana Díaz en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

En Madrid a 30 de junio de 2011

Fausto Martín López	Jesús Pedrosa Guerrero	Luis Sánchez González
DNI 70056595-K	DNI 01189268-F	DNI 53393034-Y

Agradecimientos

A mi familia y amigos , por su continuo apoyo y por estar siempre ahí, tanto en los buenos como en los malos momentos.

A mis compañeros, Jesús y Luis, que nunca se rindieron ante las dificultades, y sin los cuales este trabajo no habría sido posible.

A mis directores de proyecto, Luis y César, por su confianza, paciencia y apoyo, que han hecho posible llevar este proyecto adelante.

Gracias a todos.

Fausto Martín López

A mi familia, por apoyarme contra viento y marea sin importar cuán oscuro pintase el panorama ni cuántas trabas nos encontrásemos en el camino.

A mis amigos, por conseguir apartar mi mente de los problemas y darme perspectiva para sortear las dificultades.

A mis compañeros, Luis y Fausto, por soportarme esos días y noches de frustración y tensión cuando otros se habrían hartado hacía mucho tiempo.

Por último, que no menos importante, a mis directores de proyecto, Luis y César, por confiar en nosotros en una empresa tan nebulosa, por su paciencia en nuestras demoras y nuestra "ocasional" falta de coordinación, y por sabernos guiar en los momentos clave, anticipándose a nuestros posibles errores de planificación y la sobreestimación de nuestras posibilidades.

Sin todos ellos no sería posible que hubiéramos llegado a ninguna parte.

Muchas gracias

Jesús Pedrosa Guerrero

Agradezco a mi familia, por todo el apoyo recibido tanto en mi vida académica como en el dia a dia.

A mis amigos, por estar siempre ahí. A mis compañeros de proyecto, como no, Jesús y Fausto, porque sin ellos, este trabajo habría sido imposible de realizar. Y a mis directores de proyecto, Luis y César, por poner todo su empeño y confianza en nosotros.

Muchas gracias.

Luis Sánchez González

Resumen

Nuestro proyecto tiene como finalidad crear una aplicación libre y de código abierto que facilite el trabajo de forma colaborativa sobre diferentes editores de textos. En lugar de iniciar un proyecto desde cero, decidimos realizar nuestra herramienta sobre el editor de textos Emacs, muy popular entre programadores y usuarios técnicos, ampliando sus características para poder dotarlo de esta nueva funcionalidad.

Nuestra herramienta permite la edición de un archivo, en tiempo real, por parte de varios usuarios. Estos, a su vez, pueden hacer un seguimiento de los cambios que se van produciendo en el archivo y, de esta forma, tener una idea exacta de cómo se ha desarrollado un determinado trabajo o proyecto y de los pasos que se han ido llevando a cabo en su realización. Desde grupos de trabajo organizados en sistemas de producción altamente eficientes, como empresas, hasta pequeños conjuntos de usuarios con necesidades más modestas, todos tienen intereses cubiertos por esta herramienta. Asimismo, los proyectos que puede gestionar escalan desde ligeros textos de unas pocas líneas hasta archivos de gran tamaño. Una vez abierto un documento, todos los componentes de un grupo pueden trabajar sobre el mismo archivo y así tener acceso a las modificaciones del resto de usuarios, sin tener que esperar a que estén disponibles los demás fragmentos locales.

Otro logro importante es el de integrar un sistema de control de cambios (CVS). Este servicio se encuentra disponible mediante la instalación del cliente necesario en el ordenador, y la configuración del servidor requerido. Además, es posible acceder a este repositorio de forma normal e independiente a nuestra herramienta.

Como se ha mencionado, las herramientas utilizadas son Open Source. Esto nos permite devolver a la comunidad una aplicación libre de la que en general solo se dispone bajo pago.

Abstract

Our project aims at creating a free and open-source application that will provide collaborative editing in different popular text editors. Instead of starting a project from scratch, we decided to develop our tool based on the Emacs text editor, a popular one among programmers and technical users, by enhancing its funcionality.

Our tool allows for real time editing of a text file by multiple users. These can, in turn, track the changes being made in the file and, thus, have an exact idea of the development of a particular job or project and the steps that have been undertaken throughout its implementation. From well structured task forces in highly eficient production systems, like enterprises, to smaller sets of users with their humbler needs, they all have their interests covered by this tool. Likewise, the projects available to be managed range from the lightest texts of few lines, to more complex documents. Once a file is opened for collaborative edition, all members of a team can work on the same file, having immediate access to the work of the rest of team members.

Another important achievement is to integrate a version control system (CVS). This service is available by installing its client in the computer and setting up the proper server. The customer can also access this repository independently from our tool.

As we mentioned before, all the technologies used in the development are Open Source. This grants us an opportunity to return to the community a free and quality application that most often are only available as paid services.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

CAPÍTULO 1

Palabras Clave

Palabras clave (Castellano):

Editor, texto, colaborativo, cliente, servidor, plug-in, tiempo, real, CVS, Emacs.

Key References (English):

Editor, text, colaborative, client, server, plug-in, time, real, CVS, Emacs.

CAPÍTULO 2

Introducción

Nuestra especie se caracteriza por su gregarismo. Gustamos de la compañía de otros iguales en tanto su presencia nos permite evolucionar, avanzar, y alcanzar objetivos que, de forma individual, se tornan difíciles o costosos en extremo. Es este interés en colaborar y coordinar nuestros esfuerzos lo que nos ha llevado a desarrollar nuevos métodos de trabajo, comunicación e incluso convivencia.

El mundo en el que vivimos es, en consecuencia, mucho más pequeño gracias, en buena medida, a los avances en las tecnologías de la comunicación. Estos avances son acelerados, a su vez, por los nuevos métodos de cooperación y distribución del trabajo que tenemos. Tareas que hace tan solo unas décadas costaban meses o años de trabajo, papeleo y organización, ahora son llevadas a cabo en pocas semanas, con una eficiencia superior, unos costes muy reducidos, y un ambiente de trabajo más cohesionado y coordinado.

Es por ello que de un tiempo a esta parte, la inversión en tiempo, dinero y fuerza de trabajo se ha desviado hacia la investigación de nuevas maneras de unificar el trabajo, eliminar los esfuerzos duplicados y aligerar el desarrollo de nuevos logros.

El estado del arte en otros campos, no tan específicos como el que trata nuestro proyecto, da una idea clara de las formas en que el ser humano lucha por comunicarse de una manera más efectiva. El teléfono móvil, Internet, las tecnologías de fibra óptica, son un buen ejemplo de la dirección en que se mueve la humanidad en temas de telecomunicaciones: queremos tener más información, mejor y antes.

Pero veamos cómo hemos vivido nosotros esta evolución. Aún recordamos nuestro inicios en la carrera (e incluso antes), los trabajos conjuntos utilizando ordenadores, quedándonos en la casa de uno de los integrantes del grupo, modificando única y exclusivamente el archivo que se iba a entregar. Las actualizaciones eran lentas, el trabajo distribuido un sueño, y el concepto de *"tiempo real"* una entelequia. No eran inusuales los momentos en los que al reencontrarse los integrantes del equipo, se encontraban con que un malentendido había causado que dos o más de ellos hubiesen estado tratando el mismo punto, en lugar de diversificar sus empeños. Esto convertía los trabajos en equipo en auténticas odiseas donde la planificación se convertía en el punto de partida recurrente en cada reunión. Las herramientas tenían que cambiar, y, como suele ocurrir, el mundo avanza conforme a sus necesidades.

Decir que Internet cambió drásticamente la manera de hacer las cosas es, alternativamente, un cliché demasiado trillado, y un eufemismo de proporciones bíblicas. Simplemente cambió el mundo como si, de pronto, girase en otra dirección. Los trabajos en grupo empezaron a planificarse de manera distinta, y las reuniones en persona a espaciarse en el tiempo. El correo electrónico permitía la transmisión y compartición de archivos de forma

que, pese a tratarse todavía de un sistema basado en un único archivo, éste podía viajar a cualquier parte, a cualquier hora, y ser modificado in situ por otro componente del equipo.

Asimismo, la posibilidad de acceder a otros trabajos de terceros permitía ahorrar tiempo. Ello acercaba a los neófitos a, virtualmente, cualquier tema, al estado del arte del asunto a estudiar. Se trataba, en resumen, de subirse a los hombros de los gigantes que nos había precedido, y eso nos garantizaba una mayor facilidad para alcanzar nuevas cotas de saber.

Sin embargo, a medida que cambiábamos de siglo, nuestras herramientas, estáticas y manuales, se antojaban obsoletas. Aquello que ayer tardaba 5 minutos hoy tenía que tardar 4, lo que tenía texto solamente hoy tenía que incluir imágenes, y los conceptos que ayer volaban de un lugar a otro, hoy tenían que estar en cualquier lugar al mismo tiempo.

Las redes locales siempre habían posibilitado una manera fácil y rápida de acceder a recursos de manera rápida y segura. Los protocolos de cifrado blindaban las comunicaciones y los servicios privados. Servidores con documentos, y otras máquinas, se resguardaban de un entorno cada vez más inseguro como era Internet. Montar estas redes, sin embargo, tenía como handicap el requisito de la localidad espacial. En otras palabras, había que estar presencialmente en ese mismo lugar. Proteger comunicaciones a larga distancia, junto con el problema de la ubicuidad, que para las grandes empresas además suponía unos costes prohibitivos en espacio y equipamiento, llevó al desarrollo de una nueva forma de redes locales. Acababa de nacer la red privada virtual, o VPN por sus siglas en inglés [6].

A partir de ese momento se podía desplegar una red local con las ventajas de sencillez de uso, rapidez de montaje, mínima necesidad de mantenimiento y además, añadiéndole el valor de la deslocalización. Era un sueño para muchas empresas en aquella época, y aún hoy en grandes compañías se usa y se abusa de las VPN's. Sin embargo, la tenacidad del ser humano en su afán por evolucionar no dejó indiferente a esta tecnología tampoco.

A medida que los trabajos volvían a sobredimensionarse y su tiempo de desarrollo se dilataba, pese a las técnicas disponibles, el mundo anhelaba una nueva generación de mecanismos para solucionar la falta de tiempo. Este nuevo paso adelante fue encarnado por la tecnología de los repositorios [25].

Este nuevo conjunto de herramientas posibilitaba automatizar una tarea, tediosa como pocas, que antes solo podía realizarse manualmente: el control de versiones. Es decir, podía mantener un seguimiento, modificación a modificación, de los documentos y, poco después, además, avanzar o retroceder y actualizar o revertir cambios de los archivos que estuvieran en el repositorio. Cuando el mundo está preparado para un cambio como éste, el avance suele producirse simultáneamente en varios lugares, o en diferentes variantes, y el caso de

los repositorios no fue una excepción. Multitud de sistemas, competitivos entre ellos, en sus puntos en común y diversos en sus puntos fuertes y débiles, surgieron de la noche a la mañana para facilitar, una vez más, el empleo de equipos multiusuarios para el desarrollo de trabajos sobre ordenadores.

Alguno de estos sistemas de repositorio, como el que elegimos a la hora de implementar este proyecto, son de libre distribución. En particular el que nosotros usamos tiene que como nombre CVS [2]. Este sistema permite, como tantos otros, la creación en un servidor de una carpeta que será utilizada para almacenar los documentos con sus diferentes versiones.

No obstante, los repositorios CVS tienen un problema que supone el talón de Aquiles de toda aplicación cuyo objetivo sea la agilización de procesos colaborativos, y, es más, ese problema reside en su propio nombre. La *"concurrencia"*, ese concepto del que tan alegremente se hace eco en su título, no es una concurrencia total, sino, más bien, una automatización del mezclado del resultado de procesos de cambio paralelos. Hablando un poco más claro, lo que facilita es que dos personas modifiquen un mismo archivo a la vez, y luego mezclen sus cambios en un único documento. No existe el tiempo real, y en un mundo donde premia la velocidad, las plegarias para solucionar esta carencia no podían quedar sin ser escuchadas.

Estamos llegando al final del camino andado. A lo largo de esta ruta hemos favorecido la ausencia en detrimento de la presencia, hemos premiado la velocidad de despliegue y transmisión de conocimientos, y hemos ansiado la inmediatez en todo ello, y por último, se han presentado los editores colaborativos, para cumplir nuestras expectativas, al menos, de momento.

Tecnologías como las de Google Docs [13], documentos online que permiten ser modificados de forma realmente simultánea por varios usuarios, desde sus respectivos navegadores, o Google Wave [11], un compendio de servicios de chat y redes sociales, todo en tiempo real, han abierto las puertas a una nueva manera de entender la colaboración.

Mucho más cercano a nuestra línea de actuación, otras herramientas como Gobby [10], que busca algo similar a nuestro proyecto, nos podían dar una buena idea de por dónde empezar a mirar. Pero incluso antes de empezar nos encontramos con un delicado punto que tratar: las licencias. No somos, ni mucho menos, expertos en leyes y, más allá del poco contacto que hemos tenido en nuestra experiencia laboral, o en coloquios de algunos compañeros nuestros, al comenzar este proyecto no sabíamos nada. Sin embargo, a medida que entrábamos en materia, algunos términos como GPL [23] y LPGL [14] se nos cruzaban de manera más frecuente, y terminamos por aprender sus similitudes y diferencias, para así poder lidiar con tan escabroso asunto. Actualmente las licencias son todo un universo en pie de guerra, en

el que hay que andar con pies de plomo para poder esquivar las limitaciones que se nos imponen. Ese, y no otro, es el objetivo de extensos equipos de expertos en el área que las grandes compañías guardan bajo su manga. En nuestro caso, ese equipo éramos nosotros mismos, y el trabajo no era menor, debido a la gran cantidad de herramientas a nuestro alcance.

Un panorama así daba pie a una serie de tareas cuyo objetivo sería ponernos en contacto y sumergirnos en las tecnologías disponibles, incluyendo los editores que pretendíamos abarcar, y la manera de procurarnos su código y sus API's. Así pues, uno de nosotros investigó la documentación de Emacs, lo cual le permitiría entender mejor el código de Gobby que estaba a nuestra disposición. Por otra parte, otro se encargó de investigar la utilización de Gobby, cuales eran sus virtudes y sus defectos, y sobre todo, su estado de desarrollo. Así podríamos hacernos una idea de cual debía ser nuestra ambición, de manera realista, respecto al proyecto. Al tercero se le encargó la tarea de descargar y probar la librería net6 [16] que se utilizaba en Gobby, y posteriormente, con la información encontrada, los tres decidimos realizar una especificación de requisitos de software preliminar, a falta de ser validada por nuestros directores. La versión final se recoge en el apartado de objetivos iniciales.

Más adelante, y ya manos a la obra, decidimos que aquél que fuera el que más contacto había tenido con Emacs, y por lo tanto con Emacs-Lisp, el lenguaje con el que se escriben los plug-ins de dicho editor, fuera quien se encargara de comenzar a escribir un plug-in propio para dicho editor.

Por otro lado, uno de los dos restantes se dedicaría a desentrañar el funcionamiento de Gobby y encontrar qué ideas podríamos aplicar a nuestra implementación, y por último, el otro sería el encargado de realizar una primera versión del cliente.

Más adelante, entre los tres nos dispusimos a planear la arquitectura al completo. Un sistema cliente-servidor clásico, con las características específicas que necesitaríamos nosotros. Además dos de nosotros comenzamos a trabajar en la memoria, mientras uno más continuaba con la implementación.

Cerca del final, nos dedicamos todos a la parte de testeo del programa, conforme a los límites que nos habíamos impuesto, y la recogida de datos para su análisis. Esta tarea fue llevada a cabo por los tres de forma conjunta, puesto que era la mejor manera de comprobar la utilidad del programa.

Por último, nos concentramos en la memoria que ahora leen ustedes, y que consumió numerosas horas de un trabajo ímprobo. Como resultado, quedó la herramienta que ahora están contemplando.

En este inmenso mar de peces más y menos grandes, nuestro proyecto intenta buscar su hueco, haciendo brillar sus características únicas, como el tiempo real auténtico, su carácter gratuito, y su facilidad de mantenimiento. A lo largo del proyecto hemos intentado marcar la diferencia con respecto a aquellos programas que nos precedieron, haciendo uso del conocimiento atesorado en sus líneas de código, allí donde nos fueron accesibles, y de las ideas y conceptos subyacentes allí donde no lo fueron. Queda por ver cómo avanza el proyecto en el futuro, si llega a buen puerto, y cómo es tratado por los tiempos venideros.

Nosotros, por nuestra parte, nos sentimos orgullosos de haber dado este primer paso.

CAPÍTULO 3

Objetivos

Cuando se nos propuso este proyecto, se nos plantearon una serie de objetivos que servirían de guía para nuestro trabajo. Estos objetivos no pretendían ser una lista concreta de requisitos que se debieran cumplir estrictamente a la finalización del proyecto. Más bien, planteaban una visión global de lo que se intentaba obtener, ofreciendo ideas sobre cómo abordar el problema planteado de una forma general y ambiciosa. De esta forma, no se restringiría el marco del mismo, y seríamos nosotros y nuestras limitaciones de tiempo los que constreñiríamos las fronteras de los logros que finalmente se alcanzarían.

El principal objetivo era por tanto desarrollar una aplicación para trabajar conjuntamente en tiempo real sobre ficheros. Debía permitirse la edición del fichero por al menos dos personas simultáneamente, aunque intentando que este número fuera mayor para así aumentar la utilidad de la aplicación e incrementar los posibles usos de la misma.

Ésta aplicación tendría una arquitectura cliente-servidor, e idealmente sería multiplataforma. No debía crearse desde cero, sino que debería integrarse como plug-ins con editores de texto ya existentes, dando prioridad a Emacs y Gedit, al ser estos los editores utilizados más habitualmente por un amplio número de usuarios.

Un aspecto en el que se nos puso bastante énfasis fue el tiempo de respuesta, ya que se buscaba una edición colaborativa, lo más cercana posible a tiempo real. De esta forma, nuestra aplicación debía alejarse de otras ya existentes en el mercado, las cuales ofrecen la posibilidad de editar texto colaborativamente, pero presentan un retardo de varios segundos en la transmisión de los cambios entre los diferentes usuarios. Este objetivo suponía un gran desafío, no sólo por el breve tiempo de respuesta requerido (el cual implica una conexión de red con una velocidad de transmisión lo suficientemente rápida), sino también por los problemas de consistencia entre los ficheros locales de los distintos usuarios que provoca. Estos problemas, existentes desde el momento en que varias personas editan el mismo fichero desde distintos sitios, se complican con la disminución del tiempo entre cada cambio, por lo que se requieren complejos algoritmos para tratar de solventarlos de forma eficiente.

Otra de las cosas que se nos pedían era que el trabajo realizado fuera modular en el futuro. El motivo principal de esto es que los directores de proyecto eran conocedores de la elevada complejidad y gran tamaño del proyecto, así como de la importante labor de investigación que éste requería. Facilitando la extensión del proyecto, se ofrecía por tanto la posibilidad de que o bien nuestro trabajo fuera continuado por otros alumnos de la universidad en años posteriores, o bien se publicara en internet, poniéndolo por tanto a disposición de la comunidad, no sólo para su uso, sino también para que otros desarrolladores pudieran extender el proyecto y añadir nuevas funcionalidades.

Finalmente, también se nos sugirió la inclusión de la posibilidad de edición "offline" del documento compartido. Es decir, que una persona pudiera editar dicho documento sin hacer uso de nuestra aplicación, pero que los cambios realizados acabaran viéndose reflejados en el mismo. Para ello era necesario integrar el uso de un sistema de control de versiones con la aplicación. Si bien no se nos obligó a utilizar un sistema concreto, sí se nos indicó que dentro de lo posible se diera prioridad a CVS, al ser este el sistema empleado habitualmente por los directores de proyecto y para el cual se dispone de repositorios en los servidores de la UCM.

CAPÍTULO 4

Estructura General

4.1. Introducción a la tecnología de complementos

Un complemento o plug-in es un módulo de hardware o software que añade una característica o un servicio específico a un sistema más grande. En cuanto a aplicaciones, un plug-in es una aplicación que se relaciona con otra para aportarle una función nueva y generalmente muy específica.

Algunos tipos de aplicaciones de uso diario que suelen incluir plug-ins son los navegadores web. Los navegadores web más conocidos y utilizados en la actualidad como son Microsoft Internet Explorer, Mozilla Firefox o Google Chrome, disponen de un gran número de plug-ins que ofrecen servicios de lo más variado. Además, la creación y búsqueda de nuevos plug-ins es vital para conseguir un mayor número de usuarios, por lo que continuamente están sacando nuevas funcionalidades a la red.

Por otro lado, es frecuente requerir ciertos complementos que amplían las funciones de las páginas web para ver contenidos interactivos, vídeos y cosas similares. Un ejemplo muy conocido es Flash de Adobe, que es un plug-in que carga animaciones multimedia interactivas y se usa, por ejemplo, para ver vídeos.

Otra aplicación que solemos usar bastante son los reproductores de música. Algunos de ellos, permiten añadir plug-ins para reproducir formatos que no son soportados originalmente, producir efectos de sonido o video, mostrar animaciones o visualizaciones que se mueven de acuerdo a la música que se está escuchando, entre otras opciones. Como ejemplo específico de reproductor de música tenemos el Windows Media Player y Winamp, que soportan muchas de estas opciones.

4.2. Introducción a la arquitectura Cliente-Servidor

La arquitectura Cliente-Servidor es aquella en la que confluyen una serie de aplicaciones basadas en dos categorías que cumplen funciones diferentes, una requiere servicios y la otra los ofrece, pero que a la vez, pueden realizar tanto actividades en forma conjunta como independientemente. Esas dos categorías son justamente cliente y servidor.

Por tanto tenemos que el servidor es un proveedor de servicios y el cliente es quien los recibe. Ambos interactúan por un mecanismo de paso de mensajes, es decir, el cliente pregunta por un servicio y el servidor le da una respuesta. El cliente es quien inicia la comunicación, remitiendo una solicitud al servidor, y quedándose a la espera de recibir una respuesta por parte de este último. Un mismo cliente puede conectarse con varios servidores a la vez. Por otro lado, tenemos que el servidor se mantiene a la espera de recibir alguna

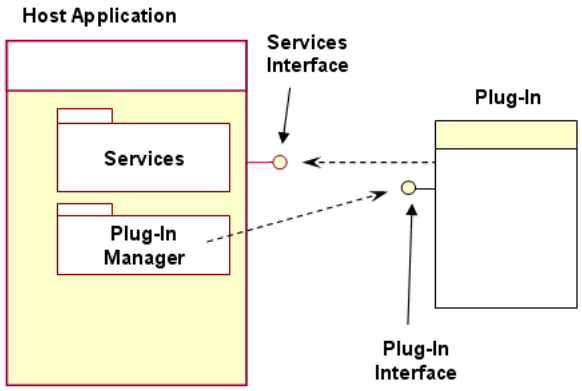


Figura 4.1: Gráfico de un plug-in

solicitud por parte de algún cliente, y una vez recibida, la procesan y envían una respuesta al cliente que la envió.

Cuando hablamos de arquitectura, nos referimos al conjunto de componentes funcionales que aprovechando diferentes estándares, convenciones, reglas y procesos, permite integrar una amplia gama de productos y servicios informáticos, de manera que pueden ser utilizados eficazmente dentro de dicha organización o conjunto. Entre las características fundamentales de esta arquitectura encontramos que tanto el cliente como el servidor pueden realizar tareas de forma conjunta como separada, ya que el cliente también tiene sus propias aplicaciones, archivos y bases de datos y que además, pueden estar en la misma plataforma o en plataformas diferentes. Por otra parte, el servidor puede ofrecer varios servicios a la vez, tanto al mismo cliente como a múltiples clientes.

El proceso servidor puede residir o no en la misma máquina que el cliente. Si no está en la misma máquina, la conexión debe realizarse a través de una red de interconexión. En este tipo de aplicaciones es necesario que el sistema operativo ofrezca servicios que permitan comunicarse a los procesos cliente y servidor. Muchas aplicaciones y servicios de red, como el correo electrónico y la transferencia de archivos, se basan en este modelo.

Esta arquitectura nos ofrece la ventaja de poder controlar los accesos de los clientes así como los recursos y los datos, por el servidor, de forma que un programa cliente que no esté autorizado para acceder a un servicio no pueda dañar el sistema. Al tener los clientes y el servidor en partes separadas, podemos aumentar su capacidad por separado, según se vayan

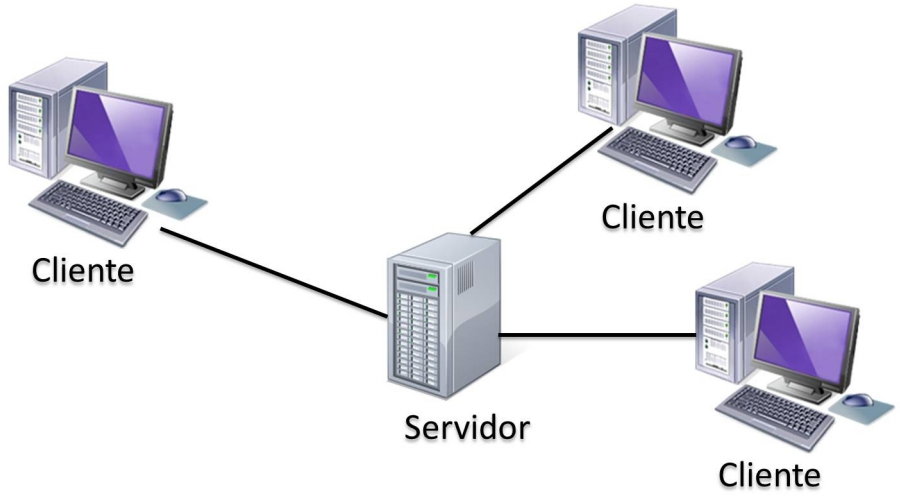


Figura 4.2: Arquitectura Cliente-Servidor

requiriendo más servicios por parte de los clientes o simplemente se requiera un aumento del número de éstos. Por esta misma razón, el mantenimiento se hace menos costoso ya que al estar distribuidas las funciones y responsabilidades entre varios ordenadores independientes, es posible reemplazar, reparar, actualizar, o incluso trasladar un servidor, mientras que sus clientes no se verán afectados por ese cambio.

Sin embargo, el uso de este modelo, puede dar problemas de congestión de tráfico, es decir, puede existir una gran cantidad de clientes que envíen peticiones en un mismo intervalo de tiempo al mismo servidor y que éste se vea saturado y tenga problemas para dar respuesta a todas las peticiones recibidas. Del mismo modo, cuando un servidor se encuentra caído, es decir, no está disponible, las peticiones de los clientes no pueden ser satisfechas. Otro de los problemas con el que nos encontramos es que para dar soporte a un gran número de clientes, el hardware y el software del servidor tiene que ser más específico de lo que normalmente nos podemos encontrar en un ordenador personal y por ello aumenta el coste en sí del servidor y todos los costes derivados de su mantenimiento y puesta a punto.

CAPÍTULO 5

Cliente

El cliente es la parte de la arquitectura dedicada a realizar peticiones de servicios al servidor y esperar que éste último se los preste por medio de una comunicación de paso de mensajes. El cliente se mantendrá a la espera de recibir las respuestas por parte del servidor. A continuación vamos a exponer dicha parte de la arquitectura Cliente-Servidor implementada, explicando los pasos que hemos llevado a cabo para su desarrollo, las versiones implementadas, así como las tecnologías estudiadas y el posible trabajo futuro.

5.1. Tecnologías estudiadas

Durante el desarrollo de las diversas versiones de los clientes tuvimos que enfrentarnos a tecnologías desconocidas para algunos de nosotros, o incluso para el grupo al completo. A continuación explicamos brevemente aquellas que pueden resultar útiles para el lector y futuros desarrolladores de cara a extender nuestro trabajo.

Sockets

Un socket es una abstracción que representa un extremo en la comunicación bidireccional entre dos procesos. Ofrece una interfaz para acceder a los servicios de red en el nivel de transporte de los protocolos TCP/IP. [21] Actualmente la interfaz de sockets está siendo estandarizada dentro de POSIX [24] y está disponible en prácticamente todos los sistemas UNIX. [12] Se puede decir que los sockets constituyen la interfaz de programación de aplicaciones más utilizada para el desarrollo de aplicaciones de Internet. Utilizando esta interfaz, dos procesos que deseen comunicarse deben crear un socket o extremo de comunicación cada uno de ellos. Cada socket se encuentra asociado a una dirección y permite enviar, o recibir, datos a través de él.

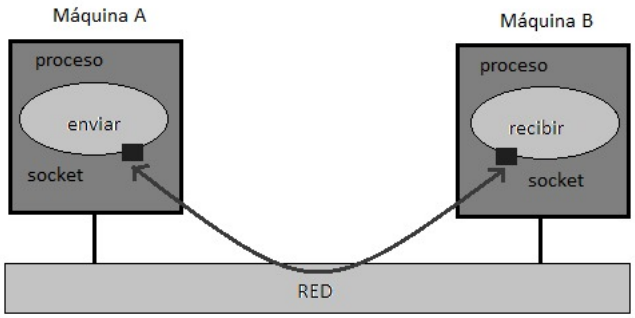


Figura 5.1: Gráfico de un socket

En nuestro desarrollo necesitábamos estas funcionalidades de cara a poder conectar cada cliente con el servidor, es decir, comunicar dos procesos que se ejecutan en máquinas distintas, para la recepción y envío de mensajes. La comunicación entre sockets es un amplio tema del cual querríamos dar unas pinceladas:

- Dominios de comunicación:

  Un dominio representa una familia de protocolos que se utiliza para el intercambio de datos entre sockets. Es importante destacar que sólo se pueden comunicar sockets del mismo dominio. En un sistema UNIX los dominios más habituales son:

  Dominio UNIX (PF_UNIX), que se utiliza para la comunicación de procesos dentro de la misma computadora.

  Dominio UNIX (PF_INET), que se emplea para la comunicación de procesos que ejecutan en computadoras conectadas por medio de los protocolos TCP/IP.

- Tipos de sockets:

  Existen dos tipos básicos de sockets que determinan el estilo de comunicación empleado. Estos dos tipos son:

  Sockets Stream (SOCK_STREAM). Con este tipo de sockets la comunicación es orientada a conexión. El intercambio de datos utilizando sockets de este tipo es fiable y además se asegura el orden en la entrega de los mensajes. El canal de comunicaciones puede verse como un flujo de bytes en el que no existe separación entre los distintos

mensajes, es decir, un proceso puede enviar un mensaje de 1KB y el proceso receptor puede recogerlo de 100 en 100 bytes. Cuando se emplea el dominio Internet, este tipo de sockets se corresponde con el protocolo de transporte orientado a conexión TCP.

Sockets de tipo datagrama (SOCK_DGRAM). Este tipo de sockets se corresponde con una comunicación no orientada a conexión. Los mensajes en este caso se denominan datagramas y tienen un tamaño máximo de 64KB. No se asegura fiabilidad, los datagramas se pueden perder y tampoco se asegura la entrega ordenada de los mismos. En este caso si existe separación entre cada uno de los distintos datagramas. Cuando se emplea el dominio Internet, los sockets de este tipo permiten acceder a los servicios del protocolo de transporte UDP.

El empleo de datagramas es, en general, más eficiente puesto que no hay etapa previa de conexión. Sin embargo, el desarrollo de los programas se complica debido fundamentalmente a los problemas de fiabilidad que presentan.

- Direcciones de sockets:

  Cada socket debe tener asignada una dirección única. Se usan para asignar una dirección a un socket local y para especificar la dirección de un socket remoto con el que se desea comunicar. Cada dominio utiliza una dirección específica. Existen por tanto dos familias de direcciones, AF_UNIX para sockets del dominio UNIX, que utilizan como dirección el nombre de un archivo local, lo que representa una dirección única y AF_INET para sockets del dominio Internet que utilizan la dirección IP de una máquina y un número de puerto dentro de la máquina. El par (dirección IP, puerto) representa también una dirección única en Internet. Cuando se trata de un socket del dominio Internet, la dirección incluye la dirección IP y un número de puerto.

Algunos servicios para el uso de sockets son:

- Creación de un socket: Un proceso puede crear un socket utilizando el siguiente servicio:

  int socket(int dominio, int tipo, int protocolo) dominio: PF_UNIX, PF_INET tipo: SOCK_STREAM, SOCK_DGRAM protocolo: generalmente a 0.

- Asignación de dirección:

  int bind(int socket, struct sockaddr *dir, int long)

- Solicitud de conexión: La solicitud de conexión es una etapa que realiza el cliente para establecer la conexión con un proceso servidor cuando se utilizan sockets de tipo stream.

int connect(int socket, struct sockaddr *dir, int long)

- Aceptación de conexiones: Ésta es una fase que realiza el servidor cuando se utilizan sockets de tipo stream. Se compone de dos etapas, la primera prepara la aceptación de conexiones:

  int listen(int socket, int backlog)

  backlog: número máximo de peticiones pendientes de aceptar que se encolarán y en la segunda se aceptan las mismas:

  int accept(int socket, struct sockaddr *dir, int *long)

- Transferencia de datos en sockets de tipo stream. Para el envío de datos:

  int write(int socket, char *mensaje, int longitud)

  int send(int socket, char *mensaje, int longitud, int flags)

  Para la recepción:

  int read(int socket, char *mensaje, int longitud)

  int recv(int socket, char *mensaje, int longitud, int flags)

Virtual Private Network

Para el funcionamiento de la aplicación, necesitamos una red privada para el paso de información, en nuestro caso, texto, entre los clientes de nuestra aplicación, de tal forma que se garantice la autenticación y la integridad de la información tratada. Es por ello que hemos optado por crear una VPN.

Una VPN es una tecnología que permite la extensión de una red pública a un espacio de red local. En una VPN, aunque utiliza una red pública, como es la de conexión a Internet, los datos son transmitidos por un canal privado, de forma que se asegura la seguridad y la integridad de la información. Para hacer posible esta seguridad, las VPN requieren de la autentificación de los usuarios, teniendo estos últimos, que verificar su identidad y así restringir el acceso al resto de usuarios no autorizados. Al hacer uso de una red pública, los datos han de ser codificados previamente para garantizar su integridad. Para ello, se hace uso de algoritmos de cifrado como Data Encryption Standard, [3] Triple DES [17] o Advanced Encrypion Standard [1].

La elección de este tipo de red privada, se daba a su seguridad a la hora de transmitir datos, como se ha explicado anteriormente. También hemos optado por crear una red privada de éste tipo para hacer las correspondientes pruebas de funcionamiento de la aplicación.

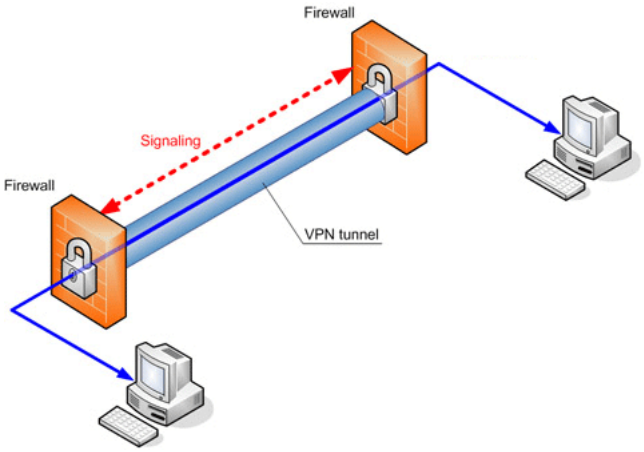


Figura 5.2: Virtual Private Network

5.2. Errores cometidos

A la hora de utilizar sockets de la librería de C, el principal error con el que nos hemos encontrado ha sido la lectura y escritura simultánea desde un socket cliente. Durante las pruebas de una de las versiones del cliente de nuestra aplicación, nos topamos con la situación de que, al ser el socket un descriptor de fichero, las lecturas y escrituras desde hilos diferentes tenían que hacerse con cuidado. Después de diversos intentos de arreglo, terminamos optando por utilizar el lenguaje de programación Java y su librería de sockets e hilos, que permitía una mayor homogeneidad al conjunto de la arquitectura y, además un desarrollo multiplataforma.

Para crear/usar una VPN, hemos usado OpenVPN [19]. Esta tecnología es una solución de conectividad basada en software, con protocolo SSL VPN completo [20] que implementa las capas 2 y 3 del modelo OSI de una extensión de red segura, utilizando el estándar de industria SSL/TLS. Soporta métodos flexibles de autentificación de clientes, basados en certificados y/o credenciales de usuario/password, y permite políticas de control de acceso orientadas a grupos o a usuarios usando reglas de firewall aplicadas a la interfaz virtual de la VPN. A la hora de crear nuestra propia VPN, hemos seguido los siguientes pasos de la página web oficial de documentación de Ubuntu. Para configurar el servidor fuimos siguiendo los pasos indicados en la sección 3: Setting up an OpenVPN server de la página anteriormente citada hasta el punto en el que hay que completar la ruta de la red inalámbrica en la configuración de ‘iptables‘, dónde tuvimos problemas para configurarlo y que no supimos resolver.

5.3. Versiones de la implementación

Primera versión del Cliente

El primer paso que dimos para la creación del cliente fue utilizar un programa como Netcat [15] que nos permitiera escuchar en un puerto y conectarnos a él. El primer punto fue aprender a lanzar subprocesos y a enviar información a estos, lo cual será tratado en la sección dedicada al cliente de Emacs.

Netcat permitía abrir un puerto para escucha con la orden "netcat -l dirección_ip nº_puerto", y conectarnos al mismo desde otro punto con la orden "netcat dirección_ip nº_puerto". Esto nos facilitó las primeras pruebas del código del plug-in en tanto en cuanto no tuvimos un cliente propio para conectarnos y enviarle la información a Emacs. No tardamos demasiado en poder descartar Netcat y comenzar a utilizar una nueva versión en C.

Segunda versión del Cliente

Nuestra primera implementación en C era lo que se puede decir, justo lo contrario de lo que necesitábamos. Debido a reutilización de código, en un primer momento, y para tener una versión simple a la que mandar datos, la primera aplicación "cliente" era precisamente un servidor. Éste hacía uso de las librerías de sockets de C, con la cual algunos de nosotros ya teníamos experiencia. La arquitectura inicial era bastante sencilla: un único hilo creaba un socket, ejecutaba la sentencia para mantenerse a la escucha, y una vez recibía la conexión de petición de otro cliente, escuchaba de ese socket e imprimía los cambios que recibía. La primera mejora que hubo que añadirle fue convertirlo en un programa multihilo que pudiera ser bidireccional, y enviar a través del socket los cambios que se detectasen en la entrada. Esto funcionó de manera adecuada para el proceso de depuración del plug-in de Emacs en las siguientes fases, pero estaba claro que no podía ser de ninguna manera una versión final, y ni siquiera código a reutilizar de forma usual.

Tercera versión del Cliente

La siguiente fase del proceso de creación del cliente nos llevó a una arquitectura de cliente de socket, lo cual nos obligaba a tener un servidor de algún tipo que escuchase en el puerto entrante. En un primer momento, y puesto que el del cliente era el lado por el que nos aproximábamos a la implementación, carecíamos de un servidor propio, por lo que recurrimos de nuevo a Netcat, en su función de escucha en un puerto, para hacer las pruebas del cliente. El desarrollo consistía en un hilo principal que creaba el socket conectándose al servidor en

netcat. Después de conectarnos al socket, el hilo principal se desdoblaba en dos, dedicados a leer y escribir en el socket. Parecía que el desarrollo sería análogo al del servidor, excepto en un detalle, y es que *no funcionaba*. El problema era básicamente que al escribir en el socket y luego intentar leer, el programa se bloqueaba, y los mismo ocurría en sentido contrario, es decir, al leer del socket y luego intentar escribir. Tras varios infructuosos intentos de arreglar la aplicación, supusimos que de alguna manera nuestra manera de utilizar el descriptor de fichero en la versión anterior y en esta debía haber cambiado, o bien que los clientes debían aplicar algún tipo de sistema antiinterbloqueo que los servidores no necesitaban. En cualquier caso, y puesto que el servidor comenzaba a tomar forma de manera paralela en lenguaje Java, decidimos romper con el código que llevábamos y aplicar nuestro conocimientos recién adquiridos sobre sockets en Java para conseguir un cliente más rápidamente.

Cuarta versión del Cliente

La última versión del cliente, que es la actual, tiene una gran cantidad de cambios con respecto a la anterior. Para comenzar no está programada en C, sino en Java, y para seguir comienza a tener visos de un protocolo de inicio de comunicación con el servidor del que antes carecía.

El cliente sigue teniendo similitudes con sus predecesores, como la estructura en dos hilos, el emisor y el receptor, pero con mejoras derivadas de los problemas con que nos habíamos encontrado. El primero y más evidente era el problema de los ecos. Éste era el descontrol provocado por la forma de trabajar el plug-in de emacs. La manera de proceder del plug-in era ,y es, detectar los cambios en el buffer y enviarlos a la entrada del subproceso, pero esto plantea un grave riesgo, puesto que al recibir un cambio a través de la conexión y escribirlo en el buffer esto es detectado por el plug-in y enviado de vuelta al programa para su envío al servidor, provocando un efecto de ping-pong en todos los clientes y todos los búffers. La manera de solucionar esto se tornó sencilla en su planteamiento pero complicada en su desarrollo. Decidimos crear una lista de cambios recibidos, de tal manera que al recibir un cambio por el hilo lector del socket, éste se almacenara en un vector, que sería consultado antes de cada emisión en busca de un cambio del mismo tipo. En un primer momento planteamos que se implementaría y utilizaría en forma de pila, puesto que, en principio, con desapilar la cima de la lista cuando fuera igual al cambio que se iba a emitir, debería ser suficiente.

Debido a motivos que aún estamos solucionando, posiblemente derivados de la complejidad de la programación multihilo con secciones críticas y en tiempo real, existen ciertos

problemas en ese procedimiento que no hemos conseguido solucionar sin modificar la implementación. Es por ello que el vector de ecos que tenemos se utiliza como una lista, y se recorre entero en busca de un cambio análogo al que se va a emitir. Esto, evidentemente, penaliza la eficiencia del cliente, si bien la baja complejidad algorítmica del código no convierten esta penalización en reactivo limitante para su escalabilidad. Evidentemente esta lista tuvo que ser bloqueada con una sección crítica para evitar los posibles conflictos a los que nos arriesgaríamos de dejarla accesible a ambos hilos.

El cliente recoge por la entrada estándar el cambio que proviene del plug-in y que contiene la información sobre el tipo de cambio, inserción o eliminación, la posición del texto de manera absoluta en que se ha llevado a cabo el cambio, el texto que se ha insertado o el tamaño que se ha eliminado en el caso de haber borrado. Para conformarlo al procedimiento de coherencia que habíamos ideado, el programa añade información sobre el identificador del cliente y el número de versión con que se envía el cambio, es decir, lo que veía el cliente en el momento de pulsar la tecla. El número de versión tiene una razón de ser clara, que el servidor pueda aplicar los cambios necesarios en el desplazamiento de los cambios venideros, pero el uso del identificador es algo ligeramente más sutil. Cuando un cliente envía un cambio al servidor, este lo remite a todos los clientes, incluido aquél que lo mandó al servidor. Sin embargo, es posible que este cambio se haya modificado en su desplazamiento, debido a cambios anteriores de otros clientes. Si bien este conflicto está contemplado en nuestro algoritmo de forma normal, existe otra posibilidad debida al acceso aleatorio y a los problemas dependientes de los retardos de envío y recepción de paquetes en las redes, temas que quedan fuera del enfoque de nuestro trabajo. Sin embargo, ideamos una manera de evitar que llegase el momento en que un cambio en un cliente produjera una discordancia entre lo que ven dos usuarios distintos. El procedimiento es conceptualmente simple: mantener una lista de cambios enviados e irla modificando como haría el servidor a medida que se reciben los cambios de los otros clientes (es decir, que tienen un identificador diferente), y en el momento en que se recibe un cambio que enviamos nosotros, comprobar que el cambio que hicimos y el que harán los demás clientes es el mismo.Este procedimiento, si bien es sencillo, por falta de tiempo no hemos terminado de implementarlo, y durante nuestras pruebas no se ha llegado a dar el caso de dos clientes viendo cambios distintos, lo que da una idea de cuán improbable es este error.

En sentido contrario, el cliente recibe un cambio proveniente del servidor que contiene la información que envió otro cliente, o incluso él mismo, y lo almacena en el vector de ecos, para su posterior anulación con el eco proveniente del plug-in. Además, recoge el número

de versión que tendremos al aplicar ese cambio, y lo actualiza en el cliente. De esta forma, cuando emitamos otro cambio, ya estaremos viendo este que hemos recibido, y lo emitiremos con el número de versión correcto.

5.4. Estado actual y posible trabajo futuro

En el estado actual el cliente es funcional en el sentido de que lleva a cabo su labor en los casos más comunes, es decir, en aquellos en los que no se dan una serie de acontecimientos relativamente poco probables. Sin embargo queda bastante trabajo por hacer de aquellos puntos que se idearon en primera instancia. La primera tarea a tratar sería la falta de un protocolo de iniciación de la comunicación completo y de un protocolo de desconexión. Actualmente el único procedimiento de conexión se compone del envío por parte del servidor, y su recepción y almacenamiento por parte del cliente, del número de identificación del cliente recién conectado, y el posterior vuelco de los cambios realizados desde que se inició la sesión del servidor. Se necesitaría poder saber si el cliente que se conecta lo hace con ninguno, todos, o alguno de los cambios que están almacenados en el servidor. Esto no ha sido posible implementarlo, aunque nuestras primeras charlas sobre el tema han barajado diversas opciones. El uso de herramientas como diff o análogas para la detección de diferencias entre archivos, el envío del archivo entero desde el servidor, o el almacenamiento en el servidor de algún tipo de identificación del cliente, podría llevar a un seguimiento detallado de la actividad de cada cliente, incluyendo desconexiones y reconexiones. Con respecto al protocolo de desconexión, en este momento no hay ninguno, aunque el código implementado permite una admisión de un protocolo sencillo, basado en objetos para cada cliente, y una organización numérica para la identificación de cada elemento de los clientes.

Por otra parte, otro punto a terminar de incluir sería el soporte para un repositorio CVS. Si bien el código está implementado para una versión temprana del servidor, y no debería costar mucho la reutilización de esas líneas para la versión actual, sí es cierto que la falta de tiempo nos ha obligado a priorizar las tareas a llevar a cabo, y ésta ha sido una de las que han tenido que quedar fuera de la implementación final.

En los clientes además debería terminar de implementarse la otra parte del algoritmo de coherencia, como segunda capa de seguridad, para evitar que los problemas de paquetes perdidos o de retrasos en los envíos pudieran dar como consecuencia una actualización diferente de los bufferes de los clientes.

CAPÍTULO 6

Servidor

El servidor es la parte de la arquitectura dedicada a proveer a los clientes de los servicios que estos le demandan. El servidor se mantendrá a la espera de recibir las peticiones por parte de los clientes. A continuación vamos a exponer dicha parte de la arquitectura Cliente-Servidor implementada, explicando los pasos que hemos llevado a cabo para su desarrollo, las versiones implementadas, así como las tecnologías estudiadas y el posible trabajo futuro.

6.1. Tecnologías estudiadas

Concurrent Versions System (CVS)

Para mantener un registro de los cambios efectuados en los ficheros y ofrecer la posibilidad de deshacerlos en caso necesario, se ha decidido incluir la posibilidad de utilizar un sistema de control de versiones desde el propio programa. Un sistema de este tipo ofrece al usuario los siguientes beneficios:

- Mecanismo de almacenamiento de los elementos que deba gestionar.
- Posibilidad de realizar cambios sobre los elementos almacenados.
- Registro histórico de las acciones realizadas sobre los ficheros, pudiendo volver a una versión anterior de los mismos.

Además de los beneficios anteriormente citados, que nos proporciona un sistema de control de versiones, en nuestro caso, el uso de este sistema ofrece una nueva funcionalidad, ya que posibilita la edición de un documento compartido desde una máquina en la que no esté instalada nuestra aplicación, aunque obviamente en este caso no se podrá editar colaborativamente en tiempo real.

Tras estudiar distintas posibilidades, la aplicación elegida para el control de versiones ha sido CVS. Los motivos principales para esta elección son el hecho de que se trata de una aplicación popular con un uso ampliamente extendido, que se distribuye bajo licencia GPL [23] y para la cual se dispone de un repositorio en la Universidad.

CVS utiliza una arquitectura cliente-servidor, donde el servidor guarda las versiones del proyecto y los archivos del historial y los clientes se conectan a éste para obtener una copia del proyecto. De esta forma, el cliente puede trabajar sobre esa copia y más tarde guardar en el servidor la copia con los cambios realizados. El servidor se encarga de acoplar todas las copias existentes, de cada uno de los clientes. Para mantener la integridad de los archivos en caso de que dos clientes estén modificando un mismo archivo, en una línea determinada, el servidor

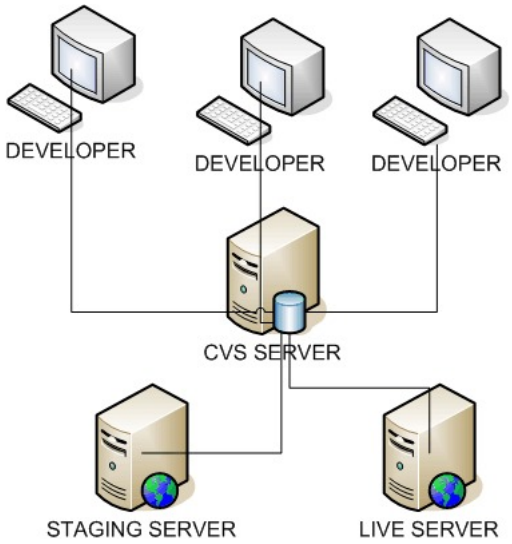


Figura 6.1: Concurrent Versions System

rechaza la segunda versión, informando al cliente del conflicto ocurrido y que tendrá que resolver a mano. Gracias al historial, un cliente puede acceder a la lista de cambios, comparar las versiones de los archivos y actualizar sus copias locales, con los archivos originales del servidor.

La forma de integrar esta aplicación con nuestro programa ha sido exigiendo que si se desea utilizar esta funcionalidad, se instale la aplicación en la misma máquina en que se encuentra el programa servidor de nuestra aplicación y se disponga de un repositorio CVS configurado adecuadamente. A partir de aquí, cuando un usuario desea realizar una acción con CVS, ejecuta el comando adecuado desde nuestro programa cliente, el cual transmite la petición al servidor. Éste bloquea a todos los clientes, indicando a los usuarios que se está realizando una operación de CVS, y una vez finalizada les avisa y desbloquea los clientes.

6.2. Errores cometidos

Durante el desarrollo de las diferentes versiones del servidor fuimos enfrentándonos a problemas de diversa índole, en muchos casos provocados por nuestra propia impericia a la hora de utilizar las herramientas que nos habíamos propuesto usar.

El uso de la librería de sockets de Java fue uno de los puntos más complicados al cuál

sobreponerse y lograr avanzar. No en vano, este es nuestro primer desarrollo sobre dicha API, y nuestra experiencia al comenzar era nula.

Uno de los errores que nos trajo más de cabeza fue el del formato de direcciones. En nuestra cabeza, las direcciones "127.0.0.1" y "locahost" eran la misma, y por ello, nuestra sorpresa fue mayúscula cuando, a la mitad del desarrollo, cuando comenzábamos las pruebas del servidor con el netcat, a éste último le era imposible conectarse como cliente a nuestro recién nacido servidor. La razón, como todas las cosas, era obvia después de ser explicada. Java toma como la dirección localhost el formato de Ipv6, es decir, en este caso concreto 0:0:0:0:0:0:0:1, mientras que netcat la nombra como 127.0.0.1. Esta incongruencia, que en un principio podía ser fácilmente obviada, nos costó dos días de búsquedas en internet y consultas a gente con mayores conocimientos.

Otro de los problemas con el cual nos vimos las caras fue el relativo al bloqueo de los sockets. Llegados a cierto punto del desarrollo, nos vimos embarcados en la búsqueda de una manera de reducir el coste en consumo de procesador de nuestro servidor. Esta tarea nos condujo a idear formas de reducir el número de hilos que manejaba el servidor, e intentamos dejarlo tan solo en dos. Los problemas vinieron cuando nos dimos cuenta de que las lecturas en los sockets de la librería estándar de Java se bloquean al leer sobre un socket vacío. Debido a las idiosincrasias de nuestro código esto era inaceptable, por lo que nos dispusimos a tratar esta situación por dos caminos, hasta descubrir cuál de los dos nos convenía más. El primero, que era el cambio de las librerías estándar de sockets de Java por la librería NIO de Java, fue descartada en una etapa temprana en favor de otro diseño, que explicamos con más detalle en las siguientes secciones.

6.3. Versiones de la implementación

Primera versión del Servidor

Nuestro primer servidor dista mucho de ser lo que ha terminado por conformar nuestro programa. La arquitectura se hacía gala de un refinamiento casi nulo, y la optimización brillaba por su ausencia.

Escrito en java, proponía un modelo basado en una ingente cantidad de hilos, que haría las veces de lectores y escritores para cada cliente, es decir, dos hilos para cada cliente. Eso suponía un uso poco eficiente de los recursos de la máquina servidora, y reducía a la mitad el número de clientes, ya como primer problema.

Sin embargo, también tiene en común algunos puntos que se llevaron de versión en versión

siendo el primero de ellos es la utilización de un lenguaje multiplataforma, que evitaba la necesidad de hacer uso de las librerías en socket de C, no disponibles para otros sistemas operativos como Windows. Si bien esto resultaría en una limitación más adelante, la lectura bloqueante, en esta primera implementación no supuso gran problema. El punto más difícil de tratar fue el control del acceso a las secciones críticas, puesto que teníamos que dar abasto para tratar los envíos simultáneos de muchos clientes, sin que eso interfiriera en el normal funcionamiento de otras partes menos congestionadas, como los hilos emisores, que solo tenían que leer de la lista los cambios ya ordenados. Aquí podían provocarse fácilmente problemas de inanición al no poseer ningún algoritmo de arbitraje para controlar el orden de acceso al recurso compartido.

En general esta versión duró poco tiempo, a medida que disponíamos de más tiempo para mejorarla y podíamos usar más y más nuestros programas cliente, sin dedicarles tanto tiempo de desarrollo.

Segunda versión del Servidor

Teniendo en cuenta la ineficiencia de nuestro primer intento, tuvimos que intentar reducir drásticamente el uso de recursos del servidor. De esta forma decidimos eliminar el uso de hilos para los clientes, y mantener solo tres hilos: el principal del servidor para la escucha en el puerto abierto a la espera de nuevos clientes, el lector, para la recepción de datos de los clientes, y el emisor para el envío de los cambios procesados.

Esta arquitectura permitía un algoritmo de lectura round robin que prometía mucho en dos sentidos. Un algoritmo round robin, selecciona todos los elementos de manera equitativa y limitando la utilización de los recursos a un período de tiempo. [18] El primero era el de eliminar los problemas de inanición al permitir un acceso secuencial a los cambios entrantes, y el segundo, algo más sutil, era el de minimizar en teoría el número de modificaciones en los cambios entrantes, puesto que la lectura se hacía de la forma más cronológica posible.

Nuestros avances teóricos fueron detenidos en seco por una realidad implacable: los sockets que estábamos utilizando eran de tipo bloqueante. Esto significaba que, imaginando un servidor con varios clientes conectados, al intentar leer secuencialmente de cada uno de ellos, tan pronto se topase con un cliente que no había escrito nada, el hilo receptor del servidor quedaría bloqueado en la operación de lectura sobre el socket, inhabilitando la lectura sobre el resto de clientes.

Como el lector podrá imaginar, esto era del todo inaceptable, y por ello sopesamos dos opciones. La primera era modificar todo el código y la arquitectura del servidor para dar uso

a la librería NIO de Java, que permite el uso sockets no bloqueantes, y la segunda era ir por la calle de en medio como se suele decir coloquialmente, es decir, tomar las mejores ideas de las dos versiones del servidor, y forjar una nueva que cumpliera los requisitos, y así es como dimos forma a la versión 3 de la implementación del servidor.

Tercera versión del Servidor

Las restricciones iniciales estaban claras: no podíamos sobrecargar la máquina servidora con dos hilos por cliente, y no podíamos permitir que el servidor se bloqueara al leer sobre un socket vacío. Así pues, la consecuencia lógica era introducir un hilo por cliente, que se bloqueara al leer sobre un socket vacío, y cuyas lecturas pasaran a introducirse en una lista de cambios, de la cual leería secuencialmente un único hilo lector del servidor, que no se bloquearía al llegar a una lista en blanco, sino que la obviaría. La implementación de esta versión introducía riesgos inherentes al uso de recursos compartidos, como eran las listas de cambios leídos de cada cliente. Es por ello que nos pusimos manos a la obra para asegurar, por medio de semáforos y secciones críticas, la coherencia dentro de las lecturas.

6.4. Estado actual y posible trabajo futuro

Lamentablemente, debido a la falta de tiempo, el trabajo paralelo sobre funcionalidades más avanzadas como el sistema de repositorio basado en cvs no ha podido ser añadido a esta versión. No obstante, se incluye el código sobre versiones anteriores del servidor, para posibilitar . Esta debería ser una tarea relativamente sencilla, puesto que se aúpa sobre los hombros de un código ya depurado y preparado para ser ampliado.

Por otro lado, el sistema de hilos y la escritura en listas puede ser encapsulado en clases ad-hoc de pilas con seguridad en los accesos para permitir la concurrencia. Asimismo existen partes del código que, si bien función conforme a los requisitos especificados, pueden ser ampliamente simplificados, debido a una reutilización de código que oscurece el mantenimiento y la modularidad.

CAPÍTULO 7

Plugins

Uno de los objetivos que nos planteamos desde el principio fue el ofrecer a los usuarios la posibilidad de utilizar nuestra aplicación con editores de texto ya existentes, más completos y pulidos que uno que pudiéramos hacer nosotros desde cero y en tan poco tiempo y con cuyo uso ya estén familiarizados. Una ventaja de esto es que de ésta forma la captación de usuarios es más fácil ya que no se .ºbliga.ª cambiar de editor, sino que permites al usuario trabajar con su editor de siempre y con todas las funcionalidades que ofrece.

Para conseguir esto, es necesario integrar nuestra aplicación con los distintos editores de texto, y la forma de conseguir esto es mediante la programación de plug-ins (complementos) específicos para cada editor que se encarguen de conectar dichos editores con los programas cliente de nuestra aplicación, transformando la información con la que trabaja nuestra aplicación a los formatos específicos de cada editor y operando con ella de forma adecuada. El primer editor para el que programamos uno de dichos plug-ins fue Emacs.

7.1. Plugin de Emacs

Tecnologías estudiadas

Un plugin de Emacs debe programarse en Emacs Lisp, un dialecto de la familia de lenguajes de programación de tipo funcional Lisp (o LISP), usado para la implementación de la mayor parte de la funcionalidad de edición de Emacs. Emacs Lisp (a veces también llamado Elisp, a pesar del riesgo de confusión con otro dialecto de Lisp con el mismo nombre) está bastante relacionado con otro dialecto de Lisp: MacLisp, y recibió cierta influencia tardía de Common Lisp. Soporta los métodos de programación imperativa y funcional.

Por tanto, dado que ninguno de nosotros conocía dicho lenguaje, el primer paso fue aprenderlo, lo cual implicó bastante tiempo y dificultad, ya que se trata de un lenguaje antiguo, bastante diferente a los lenguajes de programación más populares en la actualidad (y más vistos durante la carrera) y con un uso muy limitado, ya que aunque puede utilizarse como lenguaje de scripting, su uso se reduce principalmente a su empleo para la extensión y personalización de Emacs.

Para aprender a programar en Emacs Lisp, hemos seguido el mismo proceso de aprendizaje de cualquier lenguaje de programación, especialmente uno basado en un paradigma de programación con el que no se está familiarizado, estudiando probando diferentes comandos y funciones que nos podían ser útiles, así como empezar a probar pequeños programas para familiarizarnos con el nuevo lenguaje. Para ello nos hemos ayudado de manuales de programación en Emacs Lisp, tanto oficiales, como el GNU Emacs Lisp Reference Manual [8], como

no oficiales como por ejemplo un manual de Emacs Lisp en español [9].

Otro requisito antes de empezar la creación del plugin fue estudiar el funcionamiento interno de Emacs, para ver qué posibilidades nos ofrecía en cuanto a la detección de cambios en el fichero, inserción de texto, bloqueo de la entrada, etc., ya que esto afectaría directamente a nuestro programa cliente, el cual debería adaptarse a lo que nos ofreciera Emacs.

Una vez estudiado todo esto y tras la toma de diversas decisiones que definirían el funcionamiento de nuestra aplicación, se procedió a la implementación de una primera versión del plugin, que serviría para definir su estructura, ya que sobre ella se basarían las futuras versiones, sin realizar grandes cambios en la estructura principal. El formato de este plugin es un archivo con extensión .el que podemos cargar dinámicamente desde Emacs.

7.1.1. Funcionamiento

A grandes rasgos, el funcionamiento del plugin es el siguiente:

Al arrancar el plugin, éste ejecuta el cliente de nuestra aplicación como subproceso en segundo plano y asocia la instancia de Emacs con el fichero que le indiquemos que queremos editar colaborativamente. A partir de aquí el plugin y el cliente comenzarán a intercambiar mensajes para comunicarse. El comportamiento del editor dependerá por tanto de la entrada del usuario y de los mensajes que reciba del programa cliente de nuestra aplicación. Estos mensajes tienen una estructura definida, e incluyen entre otras cosas, el tipo de mensaje, la operación que representan y los parámetros necesarios para poder llevar a cabo dicha operación.

Cada vez que el usuario efectúa un cambio en el fichero o decide realizar un operación (como uso del sistema CVS o comprobación de consistencia), se notifica mediante un mensaje al subproceso cliente, que se encargará de enviarlo al programa servidor, el cual notificará los cambios al resto de usuarios. De igual forma, cuando se recibe un mensaje desde el subproceso cliente indicando algún cambio en el fichero por parte de otro usuario, se aplican dichos cambios inmediatamente.

La decisión de realizar y transmitir estos cambios inmediatamente se tomó con la intención de reducir la latencia y dar la sensación al usuario de una edición en completo tiempo real, aun a costa de requerir una conexión rápida entre los usuarios y de dificultar el mantenimiento de la consistencia entre los ficheros de los distintos usuarios.

7.1.2. Errores cometidos

Tras la implementación de la primera versión del plugin se detectaron los siguientes errores:

- Inserción de grandes bloques de texto al utilizar la función de pegado de texto previamente copiado o cortado.

- Edición remota de un fichero distinto al que se desea editar colaborativamente, esto es, tras iniciar una sesión de edición colaborativa, si el usuario pasaba a editar en la misma instancia de Emacs otro archivo de forma local sin cerrar la sesión, y llegaban cambios en el fichero compartido, éstos se aplicaban sin embargo sobre el fichero local.

Estos errores fueron solucionados en la segunda versión del plugin.

7.1.3. Versiones de la implementación

Primera versión

Después de diversas pruebas y la toma de contacto con el nuevo lenguaje Emacs Lisp, buscábamos tener una base de lo que sería el plugin, para ir trabajando sobre ella y realizar las futuras versiones del plugin.

El objetivo principal de esta versionó era familiarizarnos con el editor de texto Emacs y su lenguaje de programación Emacs Lisp, y definir una estructura básica del plugin para tener una base sobre la que ir asentando las próximas versiones.

Principalmente buscábamos lo que es la función principal de nuestro proyecto, la posibilidad de comunicarse e intercambiar información, así como trabajar en tiempo real con otros usuarios.

Por tanto, en esta versión, se ha conseguido comunicar dos instancias de Emacs, con la posibilidad de envío y recepción de información por ambas partes, y se han implementado simplemente las operaciones básicas (y elementales) de un editor de texto: la inserción y borrado de texto. Para la comunicación entre las distintas instancias de Emacs, se ha hecho uso del programa netcat.

Segunda versión

El motivo principal de esta versión era el corregir los problemas anteriormente mencionados. Adicionalmente también se optimizó parte del código y se realizó algún pequeño cambio.

Para solucionar el problema de la inserción de grandes bloques de texto, se modificaron las funciones que afectaban a la inserción de texto (tanto en lo que respecta a la detección de cambios, como a la aplicación de dichos cambios), de forma que permitieran tanto la inserción de caracteres como de grandes bloques de texto.

Para solucionar el problema de la edición de un fichero distinto al deseado, ideamos dos soluciones:

1 - Esta fue la solución aplicada inmediatamente para solventar el problema, ya que aunque limita la funcionalidad de la aplicación, para su implementación sólo fue necesario modificar el plugin de Emacs, con lo que no afectaba al resto de la aplicación. Consiste en asignar un fichero (con su correspondiente buffer) a cada instancia de Emacs que queramos utilizar para editar colaborativamente. De esta forma, cuando llegan cambios al programa cliente y este los pasa al plugin de Emacs, éstos se aplican siempre al buffer asignado. El inconveniente que presenta, es que dado que hay que asignar un fichero a la instancia de Emacs, sólo se puede editar un único fichero simultáneamente desde una instancia de Emacs.

2 - Esta solución se planteó para el futuro, con la intención de implementarla una vez el programa servidor estuviera en un estado más avanzado. Consiste en añadir a la cabecera de los mensajes intercambiados por las distintas partes de la aplicación un identificador del fichero al que hacen referencia. La desventaja que presenta es el incremento del tamaño de los mensajes, lo que supone un aumento del tráfico de datos entre los distintos usuarios, dado el elevado número de mensajes que se intercambian. Por el contrario, la ventaja que ofrece respecto a la primera solución, es que permite editar varios archivos simultáneamente utilizando una sola instancia del plugin de Emacs y del cliente de la aplicación, ya que el primero de estos se encargará de repartir los cambios entre los distintos ficheros según corresponda.

Tercera versión

Esta versión se realizó con la intención de añadir nueva funcionalidad a la aplicación con la integración del servidor de versiones CVS.

El plugin de Emacs debe ser capaz de permitir a los usuarios enviar las órdenes de CVS al servidor, que es dónde se va a ejecutar la aplicación CVS. Para ello se han añadido nuevos comandos y las funciones necesarias para que los usuarios puedan ejecutar estas acciones. De igual forma se ha implementado un bloqueo con avisos para notificar a los usuarios cuando se están ejecutando operaciones de CVS.

7.2. Plugin de Gedit

Uno de los objetivos propuestos para nuestro proyecto, era el de programar un plugin para conectar el editor de textos Gedit con los programas cliente de nuestra aplicación. La elección de Gedit se debió el hecho de que es un editor de propósito general, muy utilizado en sistemas Linux ya que es el editor predeterminado de GNOME [5]. Está distribuido bajo las condiciones de la licencia GPL (General Public License).

7.2.1. Tecnologías estudiadas

Un plugin para Gedit debe programarse en Python, un lenguaje de programación de alto nivel y multiplataforma. Es un lenguaje interpretado y soporta programación orientada a objetos, programación imperativa y funcional.

El gran problema al que nos enfrentábamos con esto, es que, al igual que ocurriría con Emacs y el lenguaje Emacs Lisp, desconocíamos por completo el lenguaje Python, y ahora no sólo nos veíamos con la necesidad de aprender el funcionamiento y cómo amoldarnos a Emacs para construir los programas cliente y servidor, y aprender también un nuevo lenguaje de programación, Emacs Lisp, sino que teníamos que empezar desde cero con otro lenguaje totalmente nuevo para nosotros, Python y su editor de textos Gedit.

7.2.2. Errores cometidos

Llegado a un punto, nos dimos cuenta de que la primera propuesta del proyecto era más ambiciosa y optimista de lo que podíamos abarcar, debido a que la base del proyecto era mucho más compleja de lo que en un principio pensábamos. Así que, una vez metidos de lleno en la complejidad del problema, decidimos centrarnos en construir el plugin de Emacs y los programas cliente y servidor con los que enlazarlo, y dejar el plugin para Gedit para una vez hubiéramos conseguido el otro.

7.2.3. Versiones de la implementación

En la toma de contacto con gedit se buscó de forma genérica una API para poder comunicarnos con los procesos internos del editor. Esto nos llevó a encontrar un interfaz de programación en C [7], lo cual, en aquél momento, carecía de explicación para su uso. Además, las reuniones con nuestros directores de proyecto dejaron claro que el sistema de acoplado con los editores debía hacerse sin necesidad de recompilación alguna. Esto era así con el objetivo de simplificar la interacción con el usuario. Por estos motivos, esa línea de desarrollo se

dejó de lado. La segunda acometida sobre gedit vino de la mano de Python, el cual permitía crear plugins e insertarlos en el editor sin compilaciones. Esa es una de las grandes ventajas de los lenguajes interpretados. Pese a que las limitaciones de tiempo, que nos obligaron a limitar nuestros objetivos, como el desarrollo de un plugin completo, tuvimos la oportunidad de vislumbrar las posibilidades de esta vía. La gran capacidad de adaptación del editor a las necesidades de los desarrolladores y la excelente comunidad de programadores que dedican sus esfuerzos a nutrirla constantemente de nuevos contenidos y manuales.

7.2.4. Estado actual y posible trabajo futuro

Dado que la integración de Gedit con nuestra aplicación se canceló y no se implementó ningún plugin, si en el futuro se desea añadir esta funcionalidad será preciso escribir el plugin desde cero, asegurándo el correcto funcionamiento del mismo con nuestra aplicación.

CAPÍTULO 8

Manuales

8.1. Manual de instalación

El primer paso antes de comenzar la instalación, es asegurarse de que se cumplen todos los requisitos para el correcto funcionamiento de la aplicación. Estos requisitos son los siguientes:

- Poseer una conexión de red entre los equipos Cliente y el Servidor.
- Máquina Virtual Java instalada, tanto en el equipo que alojará el servidor como aquellos donde se desee instalar el cliente.
- Editor de texto compatible con la aplicación instalado. El editor con el que se pretende hacer uso de la aplicación debe ser instalado por separado.
- Si quieren utilizar las opciones de CVS que ofrece la aplicación, éste debe estar instalado en el equipo que actuará de servidor y deberá haber un acceso directo al mismo en el PATH de la máquina.

Una vez se ha comprobado que se cumplen todos los requisitos, se puede comenzar la instalación, la cual se divide en dos partes:

- Instalación en la máquina servidor: se debe instalar el servidor de la aplicación en esta máquina. Para ello basta con copiar el fichero Java ejecutable en la misma.
- Instalación en la máquina cliente: Se debe llevar a cabo en cada equipo que se desee utilizar como cliente. El proceso se compone de la instalación del cliente y del plugin para el editor de textos:
  - En cuanto a la instalación del cliente, se realiza de la misma forma independientemente del editor de texto que se vaya a utilizar. Al igual que en el caso del servidor, la instalación consiste simplemente en copiar el fichero Java ejecutable en equipo. Posteriormente para iniciar la aplicación simplemente será necesario ejecutar dicho fichero.
  - La instalación del plugin si que depende del editor de texto que se pretenda utilizar y por tanto se deberá instalar el plugin apropiado. Por el momento, el único plugin disponible es el de Emacs y su instalación permite dos formas de llevarla a cabo:
    - La primera es instalando el archivo *.el en la carpeta de plugins de Emacs, de forma que se cargue automáticamente, simplificando este paso al usuario.

Por defecto en instalaciones UNIX basadas en Ubuntu, la ruta que debemos seguir para instalar el archivo *.el, es: /usr/share/emacs/23.2/lisp/emacs-lisp. De este modo quedará instalado en la biblioteca de plugins de Emacs y podrá ser cargado a gusto del usuario (por ejemplo mediante el mandato "load-library" o configurando Emacs para que lo cargue automáticamente cada vez que arranca).

- La segunda forma es copiando el archivo *.el donde deseemos y cargándolo dinámicamente desde Emacs cada vez que se desee utilizar, utilizando el mandato "load-file" de Emacs.

8.2. Manual de uso

Para empezar a hacer uso de la aplicación, abrimos el editor de textos sobre el que vayamos a trabajar y sobre el que tengamos el plugin instalado en la máquina, tal y como se ha descrito previamente en el manual de instalación. En este caso vamos a trabajar sobre Emacs.

Una vez abierto el editor, debemos escribir dos mandatos necesarios para cargar el plugin y empezar a trabajar con él. Estos mandatos se deben de introducir en la consola de comandos de Emacs, también conocida como "minibuffer". Para situarnos en la consola de comandos, tenemos que pulsar una de las dos siguientes combinaciones de teclas:

- Pulsando simultáneamente la combinación de teclas Alt y "x".
- Pulsando primero la tecla ESC y luego, tras soltar la tecla ESC, pulsando la tecla "x".

Una vez hecho esto, se activará una parte de la interfaz de Emacs llamada "minibuffer" y el cursor saltará automáticamente allí para que podamos escribir el nombre del mandato a ejecutar. Para dar más facilidades de uso al usuario, Emacs tiene la función autocompletado, esto es, si pulsamos la tecla "espacio", se autocompletará el nombre con los mandatos que comiencen igual que la parte que hemos tecleado. Una vez escrito el mandato, pulsamos "Intro" para ejecutarlo. Cada vez que queramos introducir un mandato, deberemos activar el minibuffer de la forma anteriormente descrita.

Los pasos a realizar son los siguientes:

- Ejecutar uno de los siguientes mandatos, load-library o load-file:
  - load-library: Este mandato se encarga de cargar el código del plugin a utilizar. Para introducirlo, basta con pulsar la combinación de teclas: "Alt + x", para que se active el minibuffer, tal y como hemos explicado antes, y como podemos ver en la figura 8.1.

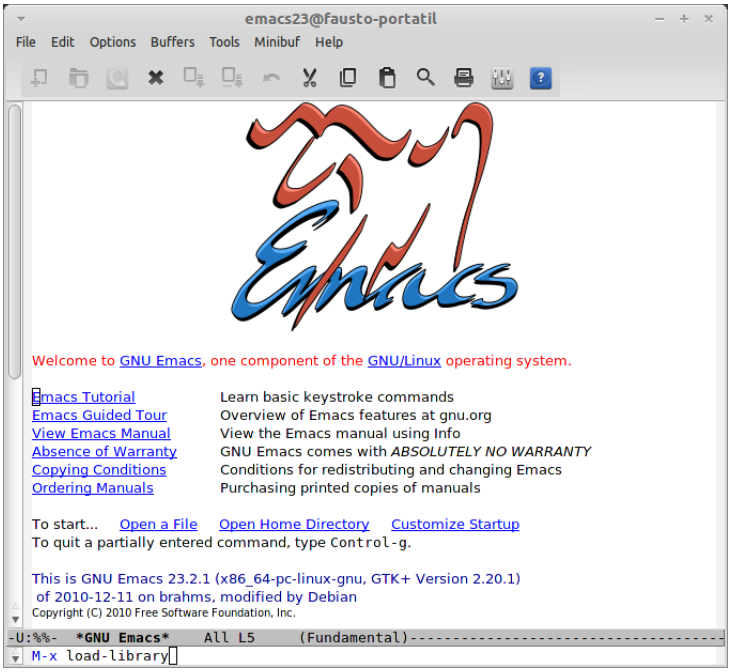


Figura 8.1: Introducir el mandato load-library

Una vez ejecutado el mandato, nos pedirá que introduzcamos el nombre del plugin, que se denomina: "edColab". Lo introducimos tal y cómo se ve en la figura 8.2 y pulsamos Intro. Hay que introducir el nombre del plugin sin la extensión del archivo. Una vez hecho esto, el plug-in estará cargado:

- load-file: Esta es la segunda opción que tenemos para cargar el plug-in. Para ello introducimos el mandato en el "minibuffer" como se explicó anteriormente. Es un mandato similar al anterior, pero que permite cargar el código de cualquier fichero fuente. Debemos utilizarlo si no hemos instalado el plugin de Emacs como se indica en el manual de instalación, mostrándole cual es la ruta del plugin en formato .el como se ve en la figura 8.3.

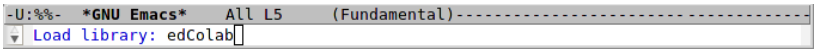
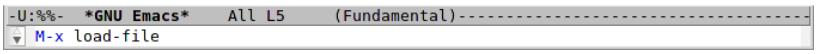


Figura 8.2: Introducir el nombre del plug-in

Introducir el mandato load-file



Indicar la ruta del plug-in

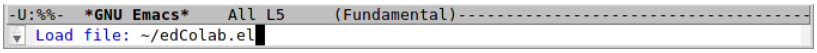


Figura 8.3: Ejecución del mandato load-file

- Abrir el fichero que queremos editar de forma colaborativa. Para ello podemos usar el mandato "find-file". Como se ve a continuación en la figura 8.4, nos pedirá el nombre del archivo, si indicamos un archivo no existente, Emacs lo creará automáticamente en blanco.

- Ejecutar el mandato "init-edColab" 8.5 con el documento a editar seleccionado. Este mandato ejecuta el cliente de nuestra aplicación como un subproceso de Emacs y configura tanto Emacs como nuestro cliente para trabajar sobre el fichero abierto por el usuario.

  A partir de este momento, ya tenemos el fichero abierto sobre el cual podremos trabajar colaborativamente con el resto de usuarios de la aplicación 8.5.

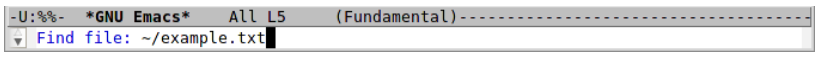
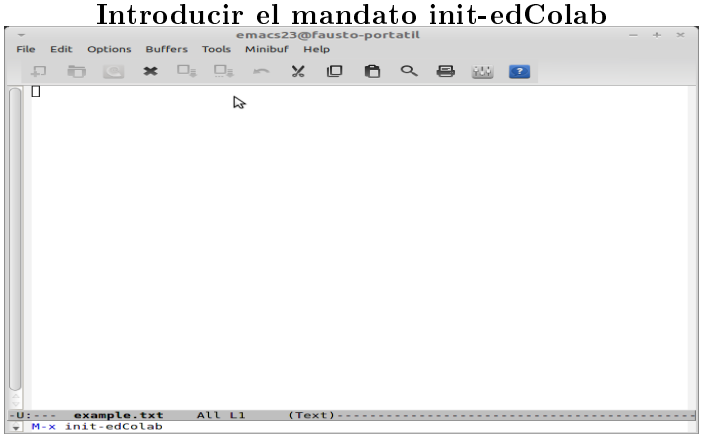


Figura 8.4: Introducir el nombre del archivo

Introducir el mandato init-edColab



Fichero abierto y listo para trabajar

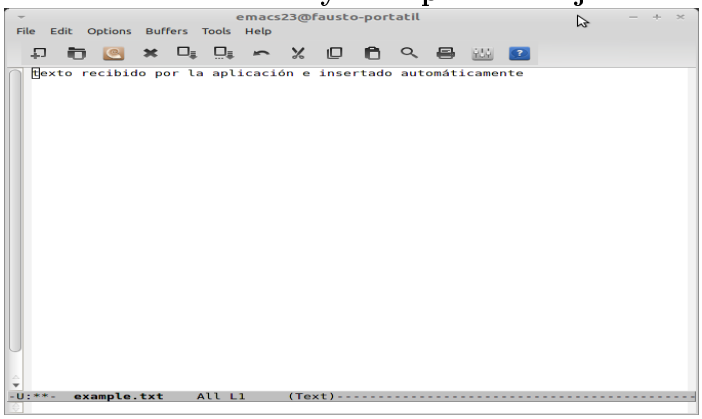


Figura 8.5: Llamada al plug-in

8.3. Manual de desarrolladores

Esta sección está dedicada a aquellos que se embarquen en la aventura de extender y ampliar nuestro proyecto.

El diagrama de clases, como se puede ver en la figura adjunta, es relativamente sencillo. El servidor, que podría suponer el mayor desafío de cara a una reimplementación, se compone básicamente de 5 clases, las cuales tiene, en su mayoría, nombres autoexplicativos. En el puesto más alto de la jerarquía se encontraría la clase "Servidor", la cual contendría, entre otras cosas, los vectores encargados de guardar la lista de entradas provenientes de los sockets, la lista de salidas que escribirían en los sockets, la lista de cambios, que almacenaría todos los cambios (o aquellos que se consideren necesarios en futuras versiones optimizadas de la herramienta) que se han recibido hasta el momento, y, por último, la lista de clientes, cuya utilidad, además de permitir el control sobre qué clientes se encuentran conectados en un instante determinado, da la posibilidad de una desconexión limpia, eliminando todos los objetos relaciones con el cliente saliente.

Por debajo de esta clase se encontraría la clase "EmisorThread", el hilo emisor del servidor, encargado de enviar a través de todas las salidas, los cambios almacenados en la lista de cambios. Su construcción en muy simple, hasta el punto de que no necesita mayor explicación.

Al mismo nivel podríamos encontrar la clase "LectorThread", la cual necesita alguna palabra más que su hermano de jerarquía. Esta clase no solo es la encargada de leer los cambios que llegan al servidor, sino que también es la responsable de modificarlos en base a los cambios que ya estaban en la lista. El algoritmo empleado se encuentra explicado en otra sección, así que evitaremos repetirlo aquí. Una vez ajustado el desplazamiento de los cambios según las circunstancias, este hilo debe insertar los cambios en la lista, bloqueándola para que el emisor no tenga la posibilidad de leer un cambio a medio hacer. Una vez introducido en la lista, el lector pierde sus funciones sobre el cambio.

También como hermano de estos dos hilos en el árbol de jerarquías podemos ver a la clase "ClienteNuevo". Es fácil ver en el código que esta clase crea un nuevo objeto conforme los clientes se conectan a nuestro servidor. Sus funciones son las de crear los bufferes de entrada y de salida particulares para ese cliente, e insertarlos en sus correspondientes listas, para hacerlos accesibles a los demás hilos. Esta clase, además, sirve para crear un hilo que actuará de separador entre en hilo lector y el socket, de tal manera que no se produzca un bloqueo en aquél.

Nuestro desarrollo se ha llevado a cabo principalmente en netbeans 6.9.1. Esto nos integraba la funcionalidad de todos los netbeans antiguos, con los cuales ya teníamos experiencia,

junto con las nuevas funcionalidades de acceso a repositorio. Así nos ahorramos la tarea de descargar los plugins y configurarlos.

Queremos ofrecer una guía muy básica del funcionamiento de Emacs y su lenguaje de programación Lisp, a aquellas personas interesadas en la continuación del proyecto, que al igual que nosotros, se enfrenten por vez primera a dicho lenguaje y su entorno. Se trata por tanto, de intentar dejar reflejados los aspectos y las acciones más elementales que tuvimos que conocer para poder dar nuestros primeros pasos, para que el usuario que lo requiera, pueda tener conocimiento y no caer así en los mismos errores.

8.4. Manual introductorio de Emacs

Antes de empezar con Emacs, es necesario saber que se trata de un editor que funciona mediante mandatos. Así pues, las acciones de abrir o cerrar un fichero, desplazar el cursor por el texto, insertar o borrar texto en el fichero, etc., se realizan todas mediante la ejecución de mandatos. Los mandatos de Emacs son funciones, que normalmente están escritas en el lenguaje Elisp. Para los usuarios, los mandatos no dejan de ser acciones que se realizan cuando pulsamos determinada combinación de teclas, o cuando seleccionamos una opción del menú. Por ejemplo, incluso para la inserción de una letra, se podría usar la siguiente combinación de teclas que invocaría el mandato de inserción de un carácter:

"Ctrl-u 6 a" escribirá cinco veces la letra "a". (Se ejecutará 5 veces ese mandato)

Como vemos, los mandatos tienen la siguiente estructura: Nombre del mandato, argumento1, ... , argumentoN.

Para la ejecución de los mandatos invocándolos por su nombre, debemos realizar las acciones anteriormente descritas en la sección "Manual de uso". Podemos invocar los mandatos de tres formas distintas: invocar al mandato por su nombre, indicándole a Emacs que queremos ejecutar un mandato de esa manera, pulsando en el teclado la combinación de teclas a la que cierto mandato esté asociado y por último seleccionar el mandato del menú o de la barra de herramientas de Emacs. Si queremos cancelar un mandato, debemos usar la combinación de teclas "Ctrl-g" y usándolo siempre que el cursor se encuentre en el "minibuffer".

Conviene saber que normalmente en los manuales de uso de Emacs y entre sus usuarios, para referirse a ciertas teclas de control, usan las siguientes abreviaturas:

- C: tecla Control
- ESC: tecla Escape
- TAB: tabulador
- SPC: barra espaciadora

- RET: tecla RETURN, INTRO o ENTER
- RETRO: tecla de RETROCESO
- M: tecla Meta

En la última abreviatura nos encontramos con el concepto de "tecla Meta". Dicha tecla, no es una tecla física, sino una especie de concepto o tecla virtual. Se asigna a dos teclas físicas: La tecla Alt-izquierda y la tecla Escape. La primera se usa pulsándola simultáneamente con otra tecla, mientras que Esc se usa pulsándola antes que la siguiente tecla.

En cuanto a los elementos de la pantalla de Emacs, sólo resaltaremos que como cualquier otro editor de textos, dispone de una barra de herramientas y barra de menús, el área de edición de textos, una barra de modo, que es dónde encontramos la información del fichero abierto en ese momento, su nombre, la posición del cursor y la línea, columna y el porcentaje del fichero en el que nos encontramos. Por último dispone de un área de ecos, utilizada para advertirnos de errores, resultados de ejecuciones, etc., y también dispone del “minibuffer”, que sirve para poder escribir mandatos y ejecutarlos.

Otro concepto muy útil del que tuvimos que hacer uso, es “Hook”. En Emacs se denomina “Hook” a un tipo especial de variable cuyo significado es el de contener dentro de sí una lista de tareas que deben realizarse por orden. Para ello suele usarse una función denominada “add-hook” que añade una tarea a la lista de tareas.

Hasta ahora no habíamos comentado el trabajo de Emacs sobre buffers. Se entiende por buffer una zona de memoria en la que se encuentra el texto objeto de edición. Cuando leemos un fichero, lo que hacemos es copiar su contenido a una zona de memoria, es decir, a un buffer.

En cuanto a mandatos, dejamos algunos que pueden ser de interés y que hemos utilizado en la programación del plugin para Emacs y que pensamos que es importante que dejemos constancia de su existencia y funcionamiento.

C-u n, Alt-n, Esc n : introduce como argumento del siguiente mandato, el número n. si n = vacío, entonces se interpreta como 4.

C-n : desplazamiento del cursor a la línea inferior

C-g : cancelamos un mandato

C-x u : deshacer el último mandato.

Con respecto a ventanas:

C-x 0 : borra la ventana activa

C-x 1 : borra todas las ventanas menos la activa

C-x 2 : divide la ventana horizontalmente

C-x 3 : divide la ventana verticalmente

C-M-v : Desplaza hacia abajo el texto de la próxima ventana

C-x C-f : Lee desde el disco un fichero y copia su contenido en un buffer donde podremos editarlo.

C-x d : Abre un directorio

C-x C-s : Guardar en disco el contenido del buffer

goto-char : desplazamiento al carácter indicado

goto-line : desplazamiento a la línea indicada

add-hook : añade una tarea a la lista de tareas

8.5. Mini-manual de Emacs Lisp

A continuación se expone un pequeño tutorial de Emacs Lisp, elaborado al comienzo del proyecto con el objetivo de que todos los miembros del grupo se familiarizaran con los conceptos más básicos del lenguaje. Si se desea profundizar más en el lenguaje, se puede consultar en manuales de elisp, [22], [8], [9] que hemos usado como guía.

Para evaluar expresiones (ejecutar código Emacs Lisp) se pueden utilizar los siguientes mandatos:

- Evaluar una línea (desde el inicio hasta el cursor):

  ```
  eval-last-sexp
  ```

- Evaluar el código seleccionado:

  ```
  eval-region
  ```

- Evaluar todo el fichero (buffer):

  ```
  eval-current-buffer
  ```

Para hacer comentarios en el código se utiliza ";". Todo lo que quede a su derecha en esa línea no se evalúa:

```
; Esto es un comentario.
```

Para mostrar un mensaje (en la barra de estado y el buffer *Messages*) se puede usar la función "message":

```
(message "Menjaje")
```

Se pueden emplear variables en los mensajes:

- Strings (cadenas de texto):

  ```
  (message "Mensaje con %s variable" "cadena de texto")
  ```

- Números:

  ```
  (message "Mensaje con %d variable númerica" 1)
  ```

- Caracteres en código ASCII:

```
(message "Mensaje con caracter variable: %c" 90)
```

- Listas:

```
(message "Mensaje con %S" ("lista" "de" "variables"))
```

Los números, si no se especifican decimales ("7" ó "7." en lugar de "7.0"), se consideran como enteros.

Funciones aritméticas: en forma prefija.

- Suma:

```
(+ 1 2 3 4)
```

- Resta (de izquierda a derecha):

```
(- 4 2 1) ; -> 10
```

- Multiplicación:

```
(* 3 2 4)
```

- División (entera o real dependiendo de los valores):

```
(/ 3 2) ; -> 1
(/ 3 2.0) ; -> 1.5
```

- Resto:

```
(% 3 2)
```

- Exponente:

```
(expt 2 10) ; -> 2 a la 10 -> 1024
```

Verdadero y falso:

- Los siguientes elementos equivalen a falso, el resto se considera verdadero:

  `nil`

  ==

  `() (lista vacía)`

- Para verdadero suele utilizarse:

  `t`

Existen múltiples funciones para comprobar tipos y similares. Suelen terminar con la letra "p" (de predicado). Ejemplos:

```
(integerp 1.)
(floatp 1.0)
```

Operadores binarios: and (y lógica), or (o lógica), <(menor que), >(mayor que), <= (menor o igual que), >= (mayor o igual que), = (igual), string= (comparación de cadenas de texto, es decir, si son iguales o no), string<(comparación de cadenas de texto en orden lexicográfico). Ejemplo:

```
(< 5 7) ; -> t (verdadero)
```

Comparación de tipo de datos y valor:

```
(equal 3 3.0) ; -> nil (falso, entero frente a real)
```

Definición de variables globales: El siguiente código declara las varibles a, b y c con los valores 1 2 y 3 respectivamente.

```
(setq a 1 b 2 c 3)
```

Definición de variables locales: existen dos métodos (ambos ejemplos declaran las variables a y b con los valores 1 y 2 respectivamente dentro de la cláusula "let"):

- Método 1:

```
(let (a b)
(setq a 1)
(setq a 2)
; resto de código
)
```

- Método 2:

```
(let ((a 1) (b 2))
; resto de código
)
```

Para declarar un bloque de expresiones que se ejecuten juntas se emplea la expresión "progn" (similar al uso de llaves ("{}") en otro tipo de lenguajes). Es útil cuando se emplea con otras expresiones (por ejemplo "if", visto a continuación). Ejemplo:

```
(if algo
(progn ...  ; si verdadero
)
(progn ... ; si falso (parte else)
)
)
```

Estructura de control condicional "if then else":

```
(if (< 2 3) (message "si")) ; sin parte else
(if (< 2 3) (message "si") (message "no")) ; con parte else
```

Si se pretende usar un "if then else" sin parte else, es preferible usar la función "when":

```
(when (< 2 3) (message "verdadero"))
```

Estructura de control iterativa "while": El siguiente ejemplo ejecuta el cuerpo del bucle 10 veces, mostrando un mensaje:

```
(let ((x 1))
(while (<= x 10)
```

```
(message "Mensaje %d" x)
(setq x (+ x 1))
)
)
```

Para definir listas de elementos (de tamaño no fijo) se emplea la expresión "list":

```
(list a b ...)
```

Algunas de las principales funciones para operar con listas son las siguientes:

- Obtener el primer elemento de la lista:

  ```
  (car miLista)
  ```

- Obtener el enésimo elemento de la lista (el primer elemento es el 0):

  ```
  (nth n miLista)
  ```

- Obtener el último elemento de la lista:

  ```
  (car (last miLista))
  ```

- Obtener todos los elementos menos el primero:

  ```
  (cdr miLista)
  ```

- Obtener desde el enésimo al último elemento:

  ```
  (nthcdr n miLista)
  ```

- Obtener todos menos los n últimos elementos:

  ```
  (butlast miLista n)
  ```

- Obener el tamaño de la lista:

  ```
  (length miLista)
  ```

- Añadir x al inicio de la lista:

  ```
  (cons x miLista)
  ```

- Concatenar dos listas:

  ```
  (append miLista1 miLista2)
  ```

- Extraer el primer elemento (lo elimina de la lista y lo devuelve como resultado de la función):

  ```
  (pop miLista)
  ```

- Eliminar los n últimos elementos y devolverlos como una nueva lista:

  ```
  (nbutlast miLista n)
  ```

- Reemplazar el primer elemento con x y devolverlo:

  ```
  (setcar miLista x)
  ```

- Reemplazar el resto de elementos (todos menos el primero) por x y devolverlo:

  ```
  (setcdr miLista x)
  ```

- Recorrer una lista. Existen dos métodos:

  - Mediante la función "mapcar":

    ```
    (mapcar funcion miLista)
    ```

  - Mediante las funciones "while" y "pop":

    ```
    (let (miLista)
    (setq miLista '(a b c))
    (while miLista
    (message "Elemento %s" (pop miLista))
    (sleep-for 1) ; esperar un segundo
    )
    )
    ```

Para definir funciones se utiliza la expresión "defun" con el siguiente formato:

```
(defun miFuncion (arg1 arg2 &optional arg 3 arg4 &rest...)
"Resumen de una línea."
"Más documentación si fuera necesaria."
(interactive) ; opcional
(let (localVar1 ....)
; ...
; se devuelve la última expresión
)
)
```

- "miFuncion" representa el nombre de la función, que será el que utilicemos para invocarla.
- "argN" son los valores que recibirá la función.
- "arg3" y "arg4" son opcionales (si no se indican toman el valor "nil").
- "&rest" indica un número indefinido de argumentos.
- "(interactive)" es opcional e indica si la función es un comando, es decir, si puede ser llamada de forma interactiva desde el intérprete de mandatos (Alt-x). Se le pueden añadir parámetros de la forma:

  ```
  (interactive "<codigo de una letra><mensaje>")
  ```

  Por ejemplo para pedir un número como argumento se utiliza el código "n":

  ```
  (interactive "n¿Cuántos años tiene?")
  ```

  Análogamente para pedir una cadena de texto se utiliza el código "s":

  ```
  (interactive "s¿Cuál es su nombre, por favor?")
  ```

Una vez definida una función, se puede llamar desde el código Emacs Lisp como cualquier otra, tanto si es interactiva como si no:

- (miFuncion arg1 arg2 arg3)

La función devolverá el valor de la última expresión evaluada.

En Lisp las variables y funciones son a menudo referidas como símbolos, ya que al contrario que en otros lenguajes de programación, las formas sin evaluar de las mismas siempre están disponibles, basta con preceder a la variable con el signo " ' ". Ejemplo:

```
('simbolo)
```

CAPÍTULO 9

Resultados

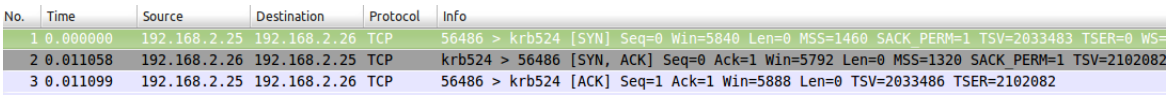


Figura 9.1: Establecimiento de la conexión

Una vez terminado el proceso de desarrollo de la aplicación, comenzamos a realizar las pruebas de funcionamiento. Para poder llevarlas a cabo de forma segura, garantizando la autenticación y la integridad de la información tratada, y poder tomar nota de los datos más relevantes, hicimos uso de una conexión VPN proporcionada por la UCM. Hemos realizado tres tipos diferentes de pruebas: una prueba conectando dos equipos con la herramienta de red, Netcat, una segunda prueba conectando dos equipos con Netcat y nuestra aplicación, y la última de ellas, haciendo conexiones a nuestro servidor de uno, dos y tres clientes, desde equipos diferentes.

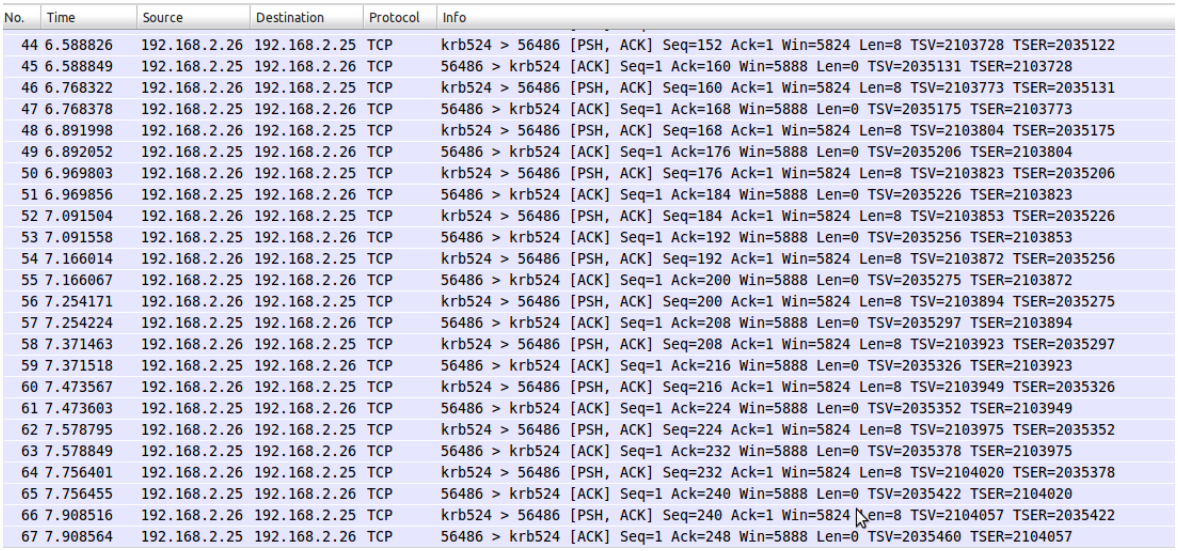
Para realizar estas pruebas, utilizamos la herramienta Wireshark [4], un analizador de paquetes de código abierto, generalmente utilizado para el análisis y solución de problemas de redes de comunicaciones y para el desarrollo de software y protocolos de comunicación, además de como herramienta didáctica. Gracias a ella podemos observar el tráfico de red generado por nuestra aplicación y estudiar el tipo y formato de los paquetes transmitidos.

La conexión entre los diferentes clientes se realiza mediante el protocolo TCP (Transmission Control Protocol, en español Protocolo de Control de Transmisión), el cual permite el flujo de datos entre los mismos. Además, TCP asegura que la comunicación se efectúe libre de errores, sin pérdidas y con seguridad. Esto permite que nuestro programa no tenga que preocuparse de problemas de pérdida o corrupción de mensajes, ya que TCP ya se encarga de eso. Algunas de las características principales de TCP de las que hace uso la herramienta y que se pueden ver a continuación (figura 9.1] en el ejemplo de sesión de edición colaborativa son las siguientes:

- Establecimiento de la conexión:

  Se realiza de la forma estándar utilizada por TCP, la negociación a tres pasos (Three-way handshake):

  - El servidor abre un socket en un puerto TCP y queda en escucha de nuevas conexiones.



No.	Time	Source	Destination	Protocol	Info
44	6.588826	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=152 Ack=1 Win=5824 Len=8 TSV=2103728 TSER=2035122
45	6.588849	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=160 Win=5888 Len=0 TSV=2035131 TSER=2103728
46	6.768322	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=160 Ack=1 Win=5824 Len=8 TSV=2103773 TSER=2035131
47	6.768378	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=168 Win=5888 Len=0 TSV=2035175 TSER=2103773
48	6.891998	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=168 Ack=1 Win=5824 Len=8 TSV=2103804 TSER=2035175
49	6.892052	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=176 Win=5888 Len=0 TSV=2035206 TSER=2103804
50	6.969803	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=176 Ack=1 Win=5824 Len=8 TSV=2103823 TSER=2035206
51	6.969856	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=184 Win=5888 Len=0 TSV=2035226 TSER=2103823
52	7.091504	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=184 Ack=1 Win=5824 Len=8 TSV=2103853 TSER=2035226
53	7.091558	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=192 Win=5888 Len=0 TSV=2035256 TSER=2103853
54	7.166014	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=192 Ack=1 Win=5824 Len=8 TSV=2103872 TSER=2035256
55	7.166067	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=200 Win=5888 Len=0 TSV=2035275 TSER=2103872
56	7.254171	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=200 Ack=1 Win=5824 Len=8 TSV=2103894 TSER=2035275
57	7.254224	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=208 Win=5888 Len=0 TSV=2035297 TSER=2103894
58	7.371463	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=208 Ack=1 Win=5824 Len=8 TSV=2103923 TSER=2035297
59	7.371518	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=216 Win=5888 Len=0 TSV=2035326 TSER=2103923
60	7.473567	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=216 Ack=1 Win=5824 Len=8 TSV=2103949 TSER=2035326
61	7.473603	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=224 Win=5888 Len=0 TSV=2035352 TSER=2103949
62	7.578795	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=224 Ack=1 Win=5824 Len=8 TSV=2103975 TSER=2035352
63	7.578849	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=232 Win=5888 Len=0 TSV=2035378 TSER=2103975
64	7.756401	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=232 Ack=1 Win=5824 Len=8 TSV=2104020 TSER=2035378
65	7.756455	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=240 Win=5888 Len=0 TSV=2035422 TSER=2104020
66	7.908516	192.168.2.26	192.168.2.25	TCP	krb524 > 56486 [PSH, ACK] Seq=240 Ack=1 Win=5824 Len=8 TSV=2104057 TSER=2035422
67	7.908564	192.168.2.25	192.168.2.26	TCP	56486 > krb524 [ACK] Seq=1 Ack=248 Win=5888 Len=0 TSV=2035460 TSER=2104057

Figura 9.2: Intercambio de mensajes

- El cliente realiza una apertura activa de un puerto mediante el paquete [SYN].
- El servidor comprueba si ese puerto está abierto, y de ser así, responde a la petición con el paquete [SYN, ACK].
- Finalmente el cliente responde con el paquete [ACK] (mensaje de confirmación).

- Envío de mensajes:

  Como se puede ver en la figura 9.2, la comunicación consiste en un intercambio de pares de paquetes [PSH, ACK] - [ACK]. Cada par de paquetes representa un mensaje, es decir, una notificación de cambios en el fichero editado.

  El paquete [PSH, ACK] lo envía el sistema que notifica los cambios, es decir, o bien el usuario que los realiza, o bien el servidor cuando se los notifica al resto de usuarios. Este paquete representa los cambios realizados, y lleva los datos generados por nuestra aplicación.

  El paquete [ACK] es un mensaje de confirmación y lo envía cada usuario que recibe un paquete del tipo anterior ([PSH, ACK]) para verificar que ha recibido el mismo correctamente.

  Algunos de los campos más importantes de estos paquetes en lo que respecta a nuestra

herramienta son los siguientes:

- Source: representa la dirección IP del usuario que envía el paquete.
- Destination: representa la dirección IP del usuario al que va destinado el paquete.
- Source port: el puerto utilizado para enviar el paquete.
- Destination port: el puerto al que va destinado el paquete. En el caso de nuestra aplicación, la instancia que crea el socket, utiliza el puerto 4444.
- Checksum: campo para la comprobación de errores. Si ha habido una corrupción del paquete durante su envío TCP se encarga de volver a transmitirlo.
- Data: este campo contiene los datos generados por nuestra aplicación, que son los que finalmente recibirán y procesarán las instancias del resto de usuarios.

Como se puede ver en la figura 9.3, el campo data contiene los mensajes de nuestra aplicación sin ningún formato ni cabecera adicional, tal y como los genera y procesa la herramienta.

Otro dato importante que se puede observar en el flujo de paquetes capturado es el campo Time (tiempo), calculado por wireshark. Se puede apreciar en figura 9.2 que los tiempos entre los distintos paquetes son bastante cortos. Esto se debe a que dado que buscábamos que la aplicación funcionase en tiempo real, simplificamos el tamaño y formato de los mensajes transmitidos, y redujimos al máximo posible el tiempo de proceso de los mismos. , En primera instancia llevamos a cabo una serie de pruebas de conexión y paso de información, sin hacer uso de nuestra aplicación, mediante la herramineta de red Netcat. Dichas pruebas consistieron en un paso de mensajes entre dos equipos conectados a la VPN. Queríamos comprobar, antes de lanzar nuestra aplicación, que el paso de mensajes era correcto y que si se producía la pérdida de alguno de ellos, éstos se volvían a reenviar y se recibían correctamente, tal y como garantiza el protocolo TCP.

Comenzamos las pruebas enviando un mensaje desde el equipo con ip 192.168.2.25 al equipo con ip 192.168.2.26 con el texto: “mensaje tcp”. Podemos ver en la figura 9.4 todo el tráfico de paquetes numerados. Comprobamos en la figura que el mensaje de prueba enviado, es el número 18. En la parte inferior podemos ver el campo Dataanteriormente mencionado. Vemos que el siguiente paquete, el número 19, es una retransmisión ya que al no recibir el emisor el paquete con el ACK, por parte del receptor, vuelve a reenviar el mensaje. El receptor envía la confirmación al emisor en el paquete número 20, dando por finalizado el envío del mensaje. Con esta prueba, pudimos comprobar que los paquetes enviados no se

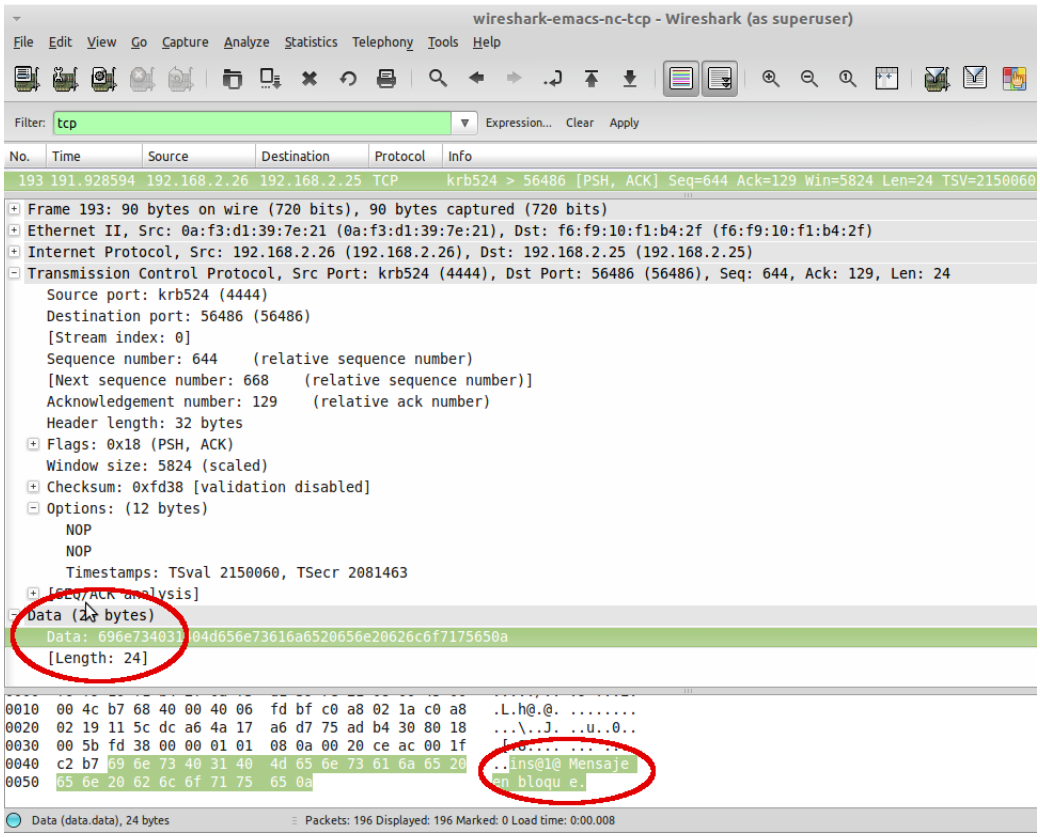


Figura 9.3: Detalle del campo Data

perdían en la comunicación ya que se volvían a retransmitir con éxito si no hubieran llegado al destino.

En la figura también podemos observar que se enviaron paquetes en ambas direcciones (paquete 35 desde 192.168.2.26 a 192.168.2.25 con mensaje: “mensaje recibido” y paquete 39 desde 192.168.2.25 a 192.168.2.26 con mensaje: “mas datos”) sin necesidad de retransmisión.

En la siguiente prueba de funcionamiento realizada, utilizamos una instancia de nuestra aplicación en uno de los extremos de la conexión y la herramienta Netcat en el otro extremo. Esta configuración es la que utilizamos en las primeras etapas del desarrollo, cuando todavía no teníamos implementada la conexión entre los clientes y nos fue de gran utilidad a la hora de depurar y ver el paso de mensajes entre los extremos de la conexión.

Comparando ambas pruebas, hemos podido observar que la diferencia de los tiempos

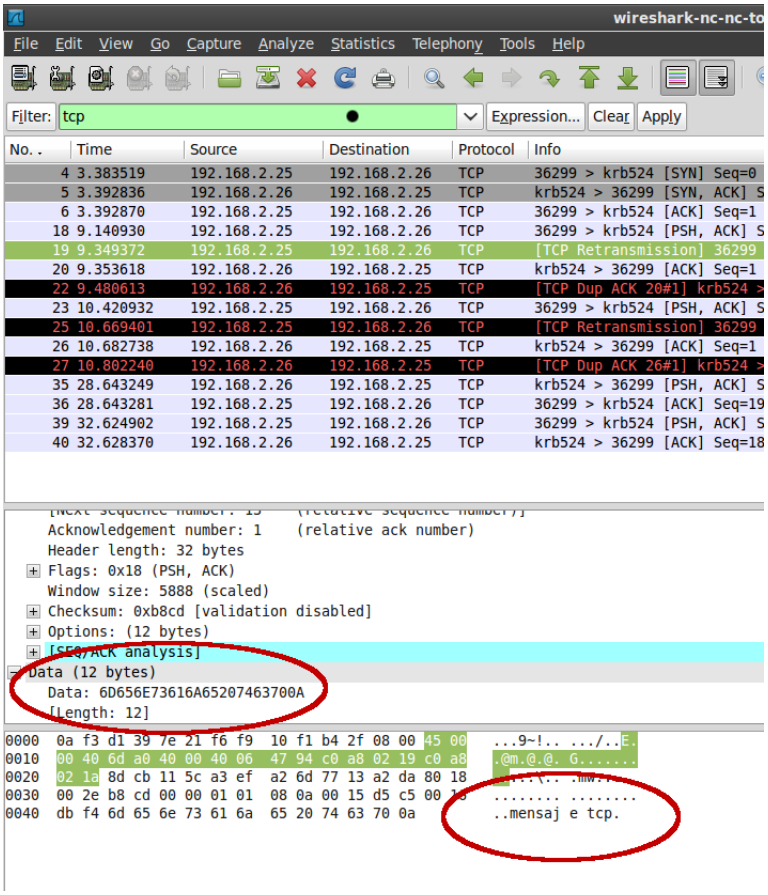


Figura 9.4: Conexión con Netcat

entre el envío de un mensaje y el siguiente es en ambos casos muy similar, situándose en tiempos inferiores a 150 - 200 milisegundos en una conexión de banda ancha estándar. Por este motivo, se puede descartar que el tiempo de proceso de los mensajes por parte de nuestra aplicación, tenga un impacto relevante en el rendimiento del flujo de la comunicación.

A continuación podemos observar nuestro programa en acción, primero con un solo usuario, y luego con dos y tres. En un primer momento compararemos el comportamiento de un escenario frente a otro, para luego pasar a compararlo con las actuaciones de Netcat y Emacs.

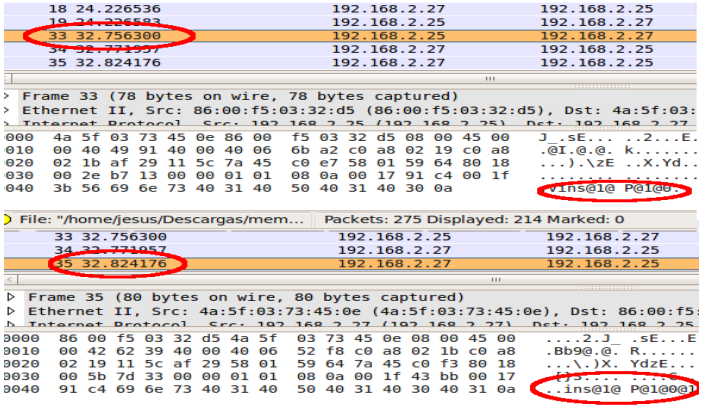
Comenzamos observando detenidamente las capturas de Wireshark. Un sencillo filtrado por paquetes TCP, que son los que transportan la información de nuestras modificaciones,

nos permite ver más claramente la transmisión de cada mensaje. En un primer momento podemos observar el protocolo desde el punto de vista del usuario que realiza la captura. Se envía el número de identificación del usuario para su posterior uso en los mensajes y, a continuación, se comienza la transmisión de los cambios.

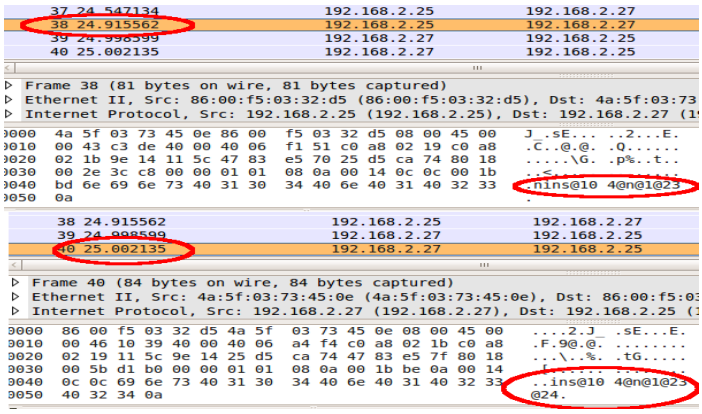
Para un único usuario está claro que lo que vemos es todo lo que ocurre en la red, sin embargo, para dos y tres, puesto que la captura está hecha desde el punto de vista de uno de los clientes, tenemos que inferir qué es lo que ocurre.

Es relativamente sencillo comprobar, como era de esperar, que el aumento de usuarios repercute en el rendimiento del programa, aunque en menor medida de lo que se pensaba en un primer momento. En teoría, podíamos pensar que aumentar en uno el número de usuarios convertiría el sistema de comprobar la lista de cambios en un problema de orden de complejidad cuadrático. Sin embargo, el hecho de limitar el número de cambios comprobados a aquellos cuya versión sea igual o posterior al mensaje entrante consigue reducir el problema drásticamente. De hecho, podemos comprobar fácilmente en las capturas de Wireshark, figura 9.5, que no hay diferencias significativas entre el tiempo de reenvío de los mensajes. La cantidad de mensajes que pueden llegar de manera concurrente al servidor es limitado, y el proceso de modificación y reordenación de los mismos es suficientemente sencillo como para que no perjudique el rendimiento.

Tiempos de viaje y retorno con un usuario



Tiempos de viaje y retorno con dos usuarios



Tiempos de viaje y retorno con tres usuarios

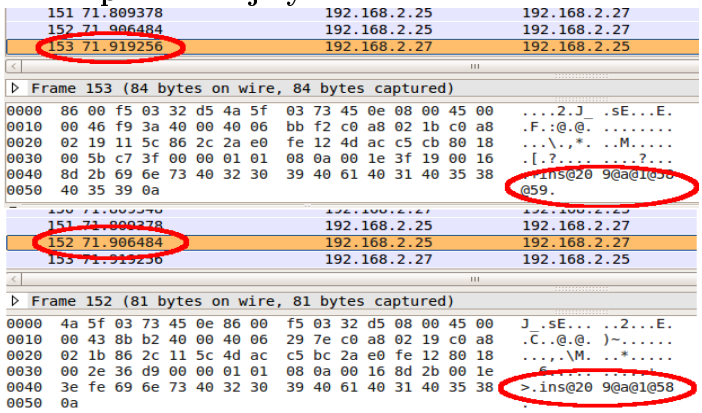


Figura 9.5: Tiempos de viaje y retorno con varios usuarios

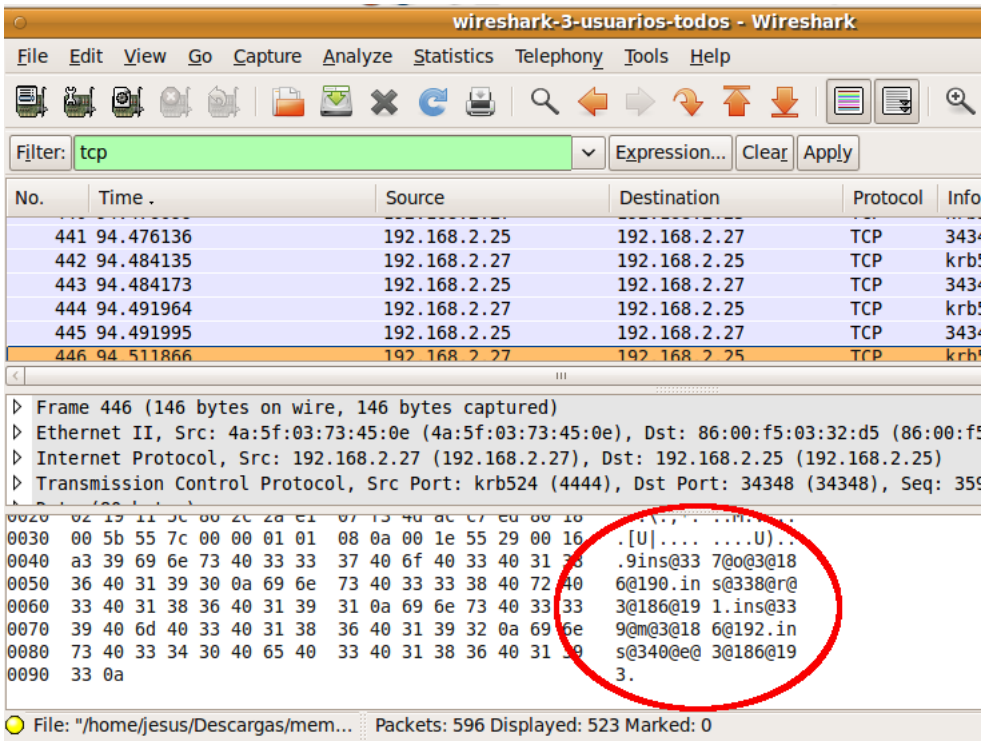


Figura 9.6: Acumulación de mensajes

Por otro lado, es interesante comprobar un comportamiento que, si bien podía reconocerse en anteriores pruebas, aquí es más patente: la acumulación de modificaciones 9.6. Por motivos de tiempo, no se pudo implementar el código necesario para asegurar la retransmisión instantánea de los datos cargados en el socket. Debido a ello, se pueden ver paquetes TCP que contienen más de una y más de dos modificaciones seguidas, lo cual, en versiones posteriores, debería eliminarse para poder permitir una fluidez mayor. Sin embargo, ese mismo problema nos permite observar que ciertamente el tamaño de un datagrama TCP sería de un provecho mucho mayor si se pudieran construir pequeños conjuntos de cambios a enviar, en lugar de uno tras otro. Más adelante comentaremos el tema con mayor detenimiento en la sección de trabajo futuro.

También atrajo nuestra atención la cantidad de paquetes perdidos y retransmitidos, visibles en la figura 9.7. Ciertamente se puede achacar a un uso externo de la misma VPN y, de hecho, en las capturas sin filtrar podemos encontrar diferentes paquetes a la dirección de

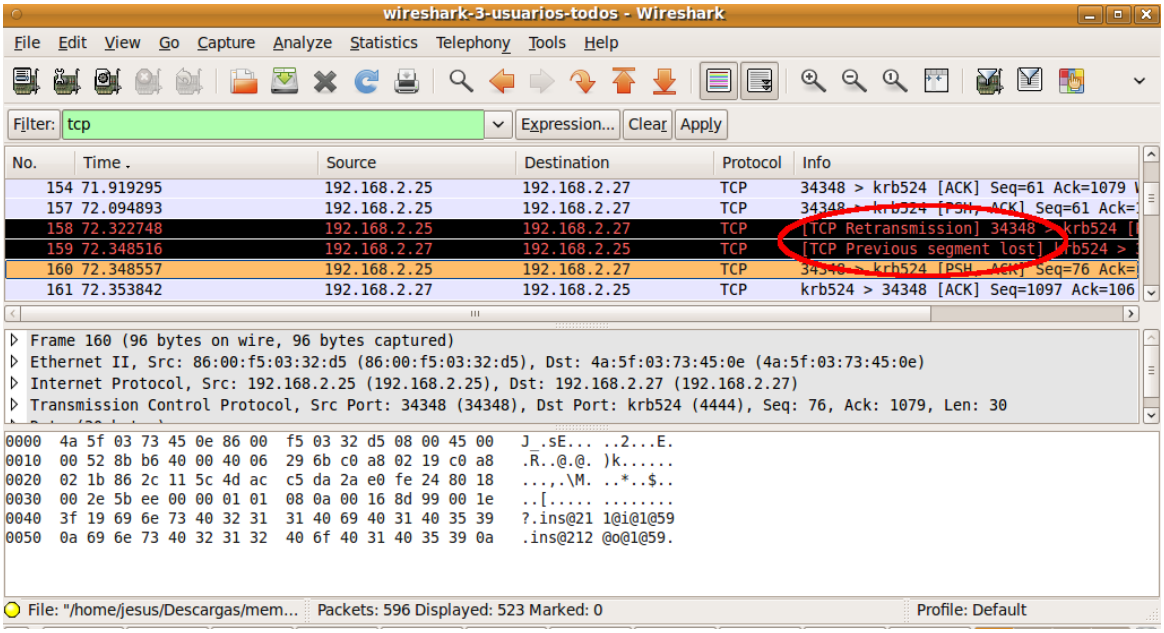


Figura 9.7: Pérdida y retransmisión de mensajes

broadcast preguntando por otras direcciones. Es posible que un escenario limpio de interferencias externas pudieran aislarse causas relacionadas con nuestro programa pero, a primera vista, las pruebas no resultan concluyentes en este sentido.

Finalmente comparamos esta última prueba, utilizando nuestra aplicación, con la anterior prueba, usando en un extremo la herramienta Netcat. Comprobamos que los tiempos de transmisión de la última prueba son sustancialmente mejores, estando en torno a los 15 - 20 milisegundos, frente a los 100 - 200 milisegundos de la segunda prueba. Se puede concluir que el incremento de tiempo en la segunda prueba, se debe al tiempo de proceso de la información por parte de la herramienta Netcat, muy elevado en comparación al tiempo de proceso combinado de nuestra aplicación, empleando los sockets de Java.

CAPÍTULO 10

Conclusiones y Trabajo Futuro

Es mejor ser aproximadamente correcto que precisamente erróneo

John Maynard Keynes

El proyecto se ha centrado en construir una herramienta que permita la edición colaborativa de un fichero. Este era el objetivo principal y se ha cumplido. Sin embargo, no ha ocurrido lo mismo con otros objetivos que se marcaron al inicio del mismo. Los motivos principales de esto fueron el marcarse unos objetivos demasiado ambiciosos y el no haber estimado correctamente el trabajo de investigación y estudio de nuevas tecnologías que el proyecto requería. Esto tuvo como consecuencia la necesidad de replantearse los objetivos a mitad del desarrollo y tener que descartar alguna de las metas menos significativas.

Pese a ello, podemos decir que los objetivos de diseño más prioritarios, los cuales consistían en que la herramienta desarrollada fuera fácilmente expandible en el futuro y que se pudiera adaptar su uso a deferentes editores de texto, sí han sido cubiertos. Esto hace que aunque el estado final de la aplicación no tiene el acabado que nos hubiera gustado conseguir, si presenta las bases necesarias para permitir la continuación del trabajo en el futuro.

Además, a pesar de que para ello ha sido necesario limitar el alcance del proyecto a un solo editor de texto, se ha conseguido implementar para la fecha de entrega una versión funcional de la aplicación utilizando Emacs. Esto supone que no sólo se puede utilizar la herramienta para realizar pruebas, sino que se puede trabajar perfectamente con ella.

Durante el desarrollo del proyecto se han presentado diversas dificultades en sus diferentes etapas. Algunas eran esperadas y habían sido planificadas de forma que pudieran solventarse de la mejor forma posible. Por el contrario, otras no habían sido tenidas en cuenta o bien su importancia había sido subestimada y supusieron que no se pudieran cumplir todos los objetivos iniciales o que se tuvieran que replantear los mismos durante el desarrollo del proyecto.

La primera de estas dificultades, era detallar los límites de la aplicación y uso, y en base a eso, comenzar a seleccionar las herramientas necesarias para llevar a cabo el proyecto. Este era un hecho determinante ya que la elección errónea de dichas herramientas, podía suponer una gran pérdida de tiempo y esfuerzo.

También se vió frenado en sus inicios el desarrollo del proyecto por la complicación que suponía el trabajar con lenguajes de programación nuevos para nosotros, en los que no teníamos ninguna experiencia. Las diferentes tecnologías implicadas y el tiempo necesario para su estudio fueron ampliamente subestimadas inicialmente, por lo que el tiempo dedicado a labores de investigación resultó ser mucho mayor de lo esperado. La consecuencia de esto fue que se hizo necesario recortar tiempo otras tareas para dedicarlo a dichas labores de investigación.

Otra de las dificultades encontradas fue adentrarse en los protocolos de sincronización de

paquetes, los cuales resultaron ser mucho más complejos de lo esperado inicialmente. Este problema fue especialmente significativo ya que se había tomado la decisión de implementar nosotros mismos dichos protocolos.

El hecho de generar, paulatinamente, la documentación de todos los aspectos ha sido, en nuestra opinión, una pieza fundamental en el buen desarrollo del proyecto.

Trabajo futuro

De aquí en adelante se abre un futuro lleno de posibilidades donde, esperamos, alguien pueda alzarse sobre nuestros hombros para alcanzar nuevas alturas. Muchas tareas de las que al inicio se pleantearon, se han quedado en el tintero. Sin embargo, lejos de tomarlo como un revés, nosotros lo vemos como una ventana a mejoras sobre lo que ya tenemos, antes de que se complique en exceso y haya que reconstruirlo desde cero.

Lo primero que nos viene a la mente de aquello que nos habría gustado implementar es un protocolo completo de conexión y desconexión que permitiera que los clientes guardasen su estado, y actualizarlo de manera individual cuando volvieran a conectarse. Esto haría mucho más sencilla la tarea de mantener vivas sesiones completas de trabajo, pese a las incorporaciones y bajas de los usuarios. Por otro lado, esto también facilitaría la posibilidad de crear grupos dentro de las sesiones, es posible que incluso asignar tareas, o permisos para ciertos párrafos.

También nos habría dejado mejor sabor de boca poder transmitir no solo texto en plano, sino también formato. Esto haría obligatorio un estudio a fondo de los sistemas de formateado en cada diferente editor. Por nuestra experiencia en asignaturas como procesadores de lenguaje, eso queda lejos de ser algo trivial, y se convierte en un trabajo hercúleo. Asimismo, la transmisión de imágenes sería algo que debería poderse implementar en futuras versiones. Hoy en día es algo casi ineludible en textos de todo tipo, desde informales hasta de investigación. A ese respecto, quizás los edtiores que hemos seleccionado estén más relacionados con el texto en sí, formateado o sin formatear, que con la edición de imágenes, pero la posibilidad de ampliarlo a otros con diferentes funcionalidades exige que tengamos visión de futuro a este respecto.

Hilando más fino, es posible que la granularidad de emisión de cambios sea optimizable mediante buffers. Nos remitimos a nuestro caso de estudio sobre las tres opciones que hemos utilizado a lo largo del proyecto para compararlas. Ahora mismo enviamos, aproximadamente, una modificación por cada tecla pulsada, lo cual, dado el tamaño de la información que enviamos y el del paquete que se forma, puede llegar a ser una pérdida de eficiencia.

Además, es posible que se de un problema sencillo de temporización: nada garantiza que los paquetes lleguen en orden, y ahora mismo ese orden no se comprueba. El uso de buffers podría acabar con ese problema, puesto que numerar bloques mayores de texto es menos costoso que numerar cada cambio. Estamos hablando de una aplicación destinada al uso masivo, y tiene que regirse por un principio fundamental muy extendido y de éxito probado: haz rápido el caso común.

Por otra parte, la integración con CVS ha venido siendo un quebradero de cabeza menor puesto que, si bien es sencillo ejecutar las órdenes desde prácticamente cualquier lenguaje de programación, el uso en tiempo real conlleva sus problemas. Tanto es así que suponemos necesario un estudio en profundidad de mecanismos de coherencia para mantener un sistema que permitiera la interrupción de actualizaciones, revertido de cambios y bloqueos.

El algoritmo de ordenación también es posible mejorarlo en ciertos aspecto. Tanto es así que algunas de las ideas que tuvimos al comienzo tuvieron que ser simplificadas por ambiciosas que eran. En un principio, los mensajes se supondrían ordenados por algún tipo de sellado de tiempo, lo que haría casi trivial la resolución antes comentada referente a los problemas de dos paquetes alcanzando el servidor en desorden. Estos métodos podrían combinarse con los ya implementados para terminar de perfilar un algoritmo capaz de hacer frente a un número de usuarios mucho más elevado, y a sistemas de redes más complejos que los puestos a prueba.

Con respecto a los editores, por nuestra mente han pasado varios posibles candidatos para futuros pasos adelante en el desarrollo. Kate, Bluefish y otros editores relacionados estrechamente con Linux, así como WinEdit, son los avances logicos, pero quizás más tarde, se podría avanzar el motor para hacerlo susceptible a textos enriquecidos como los que maneja la suite Open Office. Investigar estás posibilidades pueden hacer de este proyecto un módulo incial de algo mucho más grande. No sería complicado limitar el tipo de documento a aquél usuario con el editor más sencillo, o mostrar el código de formato en aquellos que no puedan procesarlo, abriendo así la puerta a la colaboración entre tipos diferentes de editores.

El futuro avanza rápido y, por la idiosincrasia de nuestra especie, sabemos en qué dirección. Es un buen momento para apostar por un proyecto cuya trascendencia puede que todavía no lleguemos a vislumbrar.

BIBLIOGRAFÍA

[1] Introducción al protocolo tcp ip. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[2] B. Berliner and N.B. Ruparelia. Early days of cvs. *SIGSOFT Softw. Eng. Notes*, 35:5–6, October 2010.

[3] R.F. Van Buren. How you can use the data encryption standard to encrypt your files and data bases. *SIGSAC Rev.*, 8:33–39, May 1990.

[4] G.W. Cross. Using a protocol analyzer to introduce communications protocols. In *10th ACM conference on SIG-information technology education*, SIGITE '09, pages 178–181. ACM, 2009.

[5] M. de Icaza. The gnome project: What is gnome and where is it heading? miguel tells us all. *Linux J.*, 1999, February 1999.

[6] W. Bou Diab, S. Toh, and C. Bassil. Critical vpn security analysis and new approach for securing voip communications over vpn networks. In *Proceedings of the 3rd ACM workshop on Wireless multimedia networking and performance modeling*, WMuNeP '07, pages 92–96. ACM, 2007.

[7] Gedit reference manual. `http://developer.gnome.org/gedit/3.0/`.

[8] Gnu emacs lisp reference manual. http://www.gnu.org/.

[9] Manual de emacs lisp en español. `http://gnu.manticore.es/`.

[10] Gobby a collaborative text editor. `http://gobby.0x539.de/trac/wiki/WikiStart`.

[11] M. Goderbauer, M. Goetz, A. Grosskopf, A. Meyer, and M. Weske. Syncro - concurrent editing library for google wave. In *Proceedings of the 10th international conference on Web engineering*, ICWE'10, pages 510–513. Springer-Verlag, 2010.

[12] R. Hauben. Unix and computer science. *Linux J.*, 1994, August 1994.

[13] D.R. Herrick. Google this!: using google apps for collaboration and productivity. In *Proceedings of the 37th annual ACM SIGUCCS fall conference*, SIGUCCS '09, pages 55–64. ACM, 2009.

[14] M.J. Karels. Commercializing open source software. *Queue*, 1:40:46–40:55, July 2003.

[15] The gnu netcat – official homepage. `http://netcat.sourceforge.net/`.

[16] Net6 - free software directory - free software foundation. `http://directory.fsf.org/project/net6/`.

[17] A. Parnerkar, D. Guster, and J. Herath. Secret key distribution protocol using public key cryptography. *J. Comput. Small Coll.*, 19:182–193, October 2003.

[18] R. Racu, L. Li, R. Henia, A. Hamann, and R. Ernst. Improved response time analysis of tasks scheduled under preemptive round-robin. In *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '07, pages 179–184. ACM, 2007.

[19] D. Samovskiy. Building a multisourced infrastructure using openvpn. *Linux J.*, 2008, February 2008.

[20] Secure sockets layer. `http://www.verisign.com/ssl/ssl-information-center/how-ssl-security-works/`.

[21] Introducción al protocolo tcp ip. `http://www.alcancelibre.org/staticpages/index.php/introduccion-tcp-ip`.

[22] Tutorial elisp. `http://xahlee.org/emacs/elisp.html`.

[23] Masashi Ueda. Licenses of open source software and their economic values. In *Proceedings of the 2005 Symposium on Applications and the Internet Workshops*, pages 381–383. IEEE Computer Society, 2005.

[24] S.R. Walli. The posix family of standards. *StandardView*, 3:11–17, March 1995.

[25] C.C. Williams and J.W. Spacco. Branching and merging in the repository. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 19–22. ACM, 2008.