Laboratorio Sesión 08: Sistemas de control de versiones

En esta sesión vamos a ver algo sobre los sistemas de control de versiones, muy usados en el desarrollo de software libre... y de no libre (y de hecho también para la elaboración de documentos en LaTeX :-).

8.1. CVS

El sistema `CVS` (Current Version System) es un sistema de control de versiones que se utiliza muy a menudo como ayuda para el desarrollo de aplicaciones de software libre. A pesar de que existen numerosas versiones más desarrolladas y con más prestaciones e, incluso, más bonitas ;-), el sistema de versiones de `CVS` sigue siendo muy utilizado (sí, algunos nos preguntamos por qué).

Un sistema de versiones, es, básicamente, un programa que a partir de una copia principal de un programa permite descargar copias locales y actualizarlas en el servidor central. Guarda además, un registro de cambios (de forma que estos pueden deshacerse), y detecta cuando dos o más personas han modificado las mismas líneas de un código de forma que se puede deshacer el error (aunque esto último debe hacerse a mano). En el fondo es algo muy simple pero ayuda mucho a organizar un trabajo entre varias personas (Wikipedia, en realidad, es la misma idea pero aplicada a texto).

El sistema `CVS` tiene dos partes: el servidor y los clientes. Vamos a empezar por lo fácil: crear un servidor con acceso local (bueno, primero instalad `CVS` si no está ya en el sistema).

En primer lugar deberéis decidir donde crear el directorio `CVS`. Suele ser bueno que cuelgue de `/var` (por el tema del espacio), por ejemplo en `/var/lib/cvs`. Si lo vamos a hacer así, lo primero será crear la variable global `CVSROOT` que indica el sistema de acceso al repositorio:

```
declare -x CVSROOT=/var/lib/cvs
```

Pregunta 8.1: *¿Para qué sirve el comando* `declare -x`*?*

A continuación crearemos el directorio si no existe: `mkdir -p $CVSROOT` e iniciaremos `CVS`: `cvs init`.

Con estas ordenes hemos conseguido crear la base del repositorio. En esta base NO se trabaja nunca directamente (sino `CVS` se vuelve loco) sino que se accede a través de ordenes de `CVS`. En primer lugar debemos saber como crear un módulo (un nuevo programa en desarrollo, o texto, o lo que sea...). Lo mejor es crearlo en un directorio local y a continuación enviarlo al servidor. Enviarlo es fácil, pero conviene tener mucho cuidado a la hora de decidir el esquema de ficheros ya que `CVS` no permite renombrar o mover, hay que borrar el fichero y crearlo de nuevo... y esto con un directorio entero puede ser pesado... claro que siempre nos quedan los scripts.

Id al directorio `/root/` y cread un directorio temporal. Dentro de este cread un par de ficheros vacíos (o con una cadena simple) y una vez que lo hayáis echo ejecutad:

```
cvs import <nombremodulo> <nombreorigen> <nombreversion>
```

Donde `<nombremodulo>` es el nombre del módulo (programa, texto) a crear, `<nombreorigen>` el identificador del origen del módulo y `<nombreversion>` el identificador de la versión. Por ejemplo: `cvs import prueba yomismo v1`.

Veréis que `cvs` os pide un comentario, y a continuación crea los ficheros. Ya está creado el repositorio.

A continuación ya podéis, desde cualquier parte, (en esta práctica cualquier parte significa sólo local) leer todo el módulo en un directorio (`cvs checkout <nombremodulo>`), actualizarlo (`cvs update -Pd`), enviar nuestras modificaciones locales al repositorio (`cvs commit`), o añadir y borrar ficheros (`cvs add <fichero>` y `cvs remove -f <fichero>`). Fijaros que para añadir o borrar un fichero no basta con hacerlo localmente, sino que hay que informar al repositorio de ello explícitamente y a continuación hacer el `commit`.

Además, si el proyecto consta de ficheros binarios, deberemos añadirlos de una forma especial, con la orden `cvs add -kb <fichero>` ya que sino `CVS` intentará leerlos y puede dar problemas (`CVS` no es capaz de seguir la historia de ficheros binarios, solo texto).

Probad ahora a crear un nuevo directorio temporal, a bajaros el módulo que habéis creado, a modificar los ficheros (podéis añadir alguno y borrar algún otro) y haced un `commit` de los cambios. Fijaros que cada vez os pregunta por un comentario sobre la actualización (y esto acaba siendo muy pesado).

Pregunta 8.2: *¿Qué parámetro permite incluir la línea de comentario en el comando y saltarse por tanto el paso del editor?*

Hasta ahora hemos visto como usar de forma individual `CVS`, ahora vamos a ver que pasa en caso de conflicto. Cread dos directorios temporales distintos y bajaros el módulo a ambos (si ya lo teníais en alguno no hace falta que hagáis un `checkout`, basta con que lo actualicéis con `update`). A continuación modificad el mismo fichero en la misma línea en ambos directorios y haced un `commit` de los dos. Es importante que sigáis el orden (bajar ambos temporales, modificar los dos en la misma línea y subir los dos) para que os de el aviso de conflicto detectado.

Pregunta 8.3: *¿Qué aviso de conflicto muestra* `CVS`*?*

Bien, ahora, para corregir el error, desde el temporal donde habéis obtenido el error ejecutad `cvs update` y editad de nuevo el fichero conflictivo.

Pregunta 8.4: *¿Cómo marca* `CVS` *el problema en el fichero original para que podáis corregirlo?*

Como veis el control de conflictos es bastante simple. Basta con que resolváis el conflicto a mano, dejéis una única solución valida (y borréis las marcas) y volváis a hacer `commit`. Ya está solucionado.

Otros comandos útiles que podéis mirar son: `cvs diff` (útil cuando hay conflictos) y `cvs history`, o incluso: `cvs update -D "10 minutes ago"`.

Pregunta 8.5: *¿Para qué sirve el parámetro* `-D`*?*

Tened cuidado con este parámetro porque es "pegajoso", es decir, su valor queda permanentemente en todas las ordenes siguientes hasta que lo cambiéis por `now`.

Otra opción interesante es `cvs update -r 1.2`.

Pregunta 8.6: *¿Para qué sirve esta opción? ¿Cómo se vuelve a la última versión?*

Finalmente comentaros que hay un fichero: `~/.cvsrc` que podéis crear con los parámetros que queréis que use `CVS por defecto`, por ejemplo:

```
update -Pd
diff -uw
...
```

Pues esto es básicamente `CVS`. Falta por ver como realizar el acceso al repositorio desde otra máquina. Poneros de acuerdo con vuestros compañeros (o utilizad el acceso `localhost` para simular que estáis en otro ordenador).

Básicamente hay dos formas de acceder a `CVS`, mediante `ssh` o mediante `pserver`. Son muy similares, pero `ssh` es más segura y algo más simple, así que es la que veremos aquí.

En primer lugar deberéis poner en marcha el servidor `ssh` (si estáis trabajando todo el rato como `root` aseguraros de quitar las opciones de `sshd_config` que impiden que `root` haga `ssh`). Aseguraros de que `ssh` funciona: `ssh root@localhost`. (Si trabajáis con los compañeros del ordenador de al lado, deberéis usar el nombre de su ordenador en lugar de `localhost`, claro).

A continuación deberéis cambiar la variable `CVSROOT` en la máquina cliente indicando el nuevo acceso a usar (`ext`), vuestro login y la máquina destino:

```
export CVSROOT=:ext:root@localhost:/var/lib/cvs
```

Si todo ha ido bien ya podéis acceder desde el cliente con los mismos comandos que habéis usado en local. Probadlo, pero tened en cuenta que si alguna vez montáis un servidor `CVS` no debéis dejar que la gente entre como `root`, sino que deberéis crear usuarios dedicados para leer y escribir los datos (fijaros que podéis gestionar los permisos de los ficheros de forma que ciertos usuarios puedan solo leer los datos mientras que otros puedan escribirlos).

8.2. Git

Actualmente es mucho más común usar un sistema de control de versiones más avanzado que `CVS` para gestionar los proyectos. Uno de los mejores (a mi entender) es `Git`. `Git` fue creado por Linus Torvalds que ha dicho que el nombre (en inglés viene a significar estúpido) lo puso porque: *"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'"*. La página de `man` describe `git` como *"the stupid content tracker"*.

En cualquier caso, `git` es un ejemplo de como surge y mejora el software libre. En sus inicios Linus usaba BitKeeper para desarrollar Linux. Sin embargo los propietarios decidieron retirar la licencia de libre uso debido a que, al parecer, algún ingeniero había desencriptado sus protocolos. Las reflexiones de Linus para desarrollar `Git` fueron las siguientes (transcritas de un correo suyo):

However, the SCMs I've looked at make this hard. One of the things (the main thing, in fact) I've been working at is to make that process really efficient. If it takes half a minute to apply a patch and remember the changeset boundary etc. (and quite frankly, that's fast for most SCMs around for a project the size of Linux), then a series of 250 emails (which is not unheard of at all when I sync with Andrew, for example) takes two hours. If one of the patches in the middle doesn't apply, things are bad bad bad.

Now, BK wasn't a speed demon either (actually, compared to everything else, BK is a speed demon, often by one or two orders of magnitude), and took about 10-15 seconds per email when I merged with Andrew. HOWEVER, with BK that wasn't as big of an issue, since the BK-BK merges were so easy, so I never had the slow email merges with any of the other main developers. So a patch-application-based SCM "merger" actually would need to be faster than BK is. Which is really really really hard.

So I'm writing some scripts to try to track things a whole lot faster. Initial indications are that I should be able to do it almost as quickly as I can just apply the patch, but quite frankly, I'm at most half done, and if I hit a snag maybe that's not true at all. Anyway, the reason I can do it quickly is that my scripts will not be an SCM, they'll be a very specific "log Linus' state" kind of thing. That will make the linear patch merge a lot more time-efficient, and thus possible.

(If a patch apply takes three seconds, even a big series of patches is not a problem: if I get notified within a minute or two that it failed half-way, that's fine, I can then just fix it up manually. That's why latency is critical-if I'd have to do things effectively "offline", I'd by definition not be able to fix it up when problems happen).

Con estos prerequisitos, Linux llegó a las siguientes reglas de diseño:

1. Tomar CVS como un ejemplo de lo que no había que hacer: en caso de duda hacer exactamente lo contrario. Citando de nuevo a Linus:

   For the first 10 years of kernel maintenance, we literally used tarballs and patches, which is a much superior source control management system than CVS is, but I did end up using CVS for 7 years at a commercial company (Transmeta) and I hate it with a passion. When I say I hate CVS with a passion, I have to also say that if there are any SVN (Subversion) users in the audience, you might want to leave. Because my hatred of CVS has meant that I see Subversion as being the most pointless project ever started. The slogan of Subversion for a while was "CVS done right", or something like that, and if you start with that kind of slogan, there's nowhere you can go. There is no way to do CVS right.

Como podéis ver, Linus no es un gran fan de `CVS` y todos los sistemas que de él dependen.

2. Soportar el trabajo distribuido tal y como hacía BitKeeper (de nuevo cita de Linus):

   BitKeeper was not only the first source control system that I ever felt was worth using at all, it was also the source control system that taught me why there's a point to them, and how you actually can do things. So Git in many ways, even though from a technical angle it is very very different from BitKeeper (which was another design goal, because I wanted to make it clear that it wasn't a BitKeeper clone), a lot of the flows we use with Git come directly from the flows we learned from BitKeeper.

3. El sistema debía tener un fuerte sistema de seguridad contra la corrupción de los datos, accidental o intencionada.

4. Debía dar un gran rendimiento.

El problema era que los primeros tres criterios eliminaban todos los sistemas de control de versiones ya existentes excepto "`Monotone`" y el cuarto los excluía todos. Así que Linus decidió escribir el suyo propio (que fue solo aplicable a su proyecto y poco usable con respecto a otros sistemas de control de versiones igual que Linux iba a ser un pequeño experimento que solo iba a correr en su ordenador).

Bueno, toda esta disertación sobre `Git` en realidad tiene un objetivo y es mostrar los beneficios de la filosofía de trabajo **KISS** que muchas veces vale la pena tener en cuenta.

Pregunta 8.7: *¿Qué significa* `KISS`*? ¿Qué filosofía de diseño plantea?*

La principal característica diferencial de `Git` respecto a los sistemas anteriores es que no mantiene las diferencias de unas versiones respecto a otras sino que copia todos los ficheros del repositorio (con todas las versiones) cada vez que se produce una actualización. Esto hace que sea muy fácil (y rápido) trabajar en local, e incluso, desconectado. Además, todas las copias del repositorio son en realidad el repositorio completo así que es muy fácil restaurar y/o recuperar información.

Bueno, visto todo esto vamos a ver como funciona `Git`. Primero instalaros el paquete. A continuación, al igual que en `CVS` deberéis crear un repositorio. Para ello cread un directorio vacío y entrad en él. A continuación escribid:

```
git init
```

Si todo ha ido bien esto os creará un directorio llamado `.git` que contendrá la información sobre el repositorio. A continuación podéis mirar en que estado se encuentra el repositorio (`git status`) y crear un nuevo fichero (de la forma usual).

Pregunta 8.8: *¿Qué mensaje muestra* `git status` *referente al nuevo fichero?*

El mensaje de `git status` nos dice que tenemos un nuevo fichero pero que no está en el repositorio. En `git` hay tres estados: cambios que "no existen" para `git` (untracked), cambios que están pendientes (staged) y cambios que están en el repositorio (committed). Si queremos añadir el nuevo fichero a los cambios pendientes tenemos que incluirlo con la orden:

```
git add <nombrefichero>
```

Fijaros que ahora el fichero ha pasado a estar pendiente de "envío" hacia el repositorio. Bien, si ya estamos seguros podemos incorporar el fichero al repositorio (en realidad la orden actualiza todos los cambios pendientes) con `git commit`. El problema de esta orden es que, al igual que `CVS`, nos pide un comentario con un editor de texto más o menos engorroso. Abortad (`^X`) y ejecutad el `commit` con la opción `-m` y un comentario entrecomillado.

Pregunta 8.9: *¿Para qué sirve el parámetro* `-m`*? ¿Qué información da* `git` *referente al nombre y correo del "committer"? ¿Cómo se soluciona?*

Podéis comprobar el nuevo estatus del repositorio. Probad a añadir algunos ficheros más al repositorio (individualmente o en grupo) y a continuación consultad la historia del repositorio mediante el comando `git log`. Fijaros que si modificáis algún fichero que ya habéis añadido (probad a hacerlo) `git` no guarda las modificaciones inmediatamente sino que nos avisa de que las ha habido y de que hay que hacer un nuevo `git add` para añadirlas a los cambios que serán enviados.

Pregunta 8.10: *¿Qué información muestra* `git log`*? ¿Y* `git status`*? ¿Cómo dice* `git status` *que se ha de volver a la versión sin cambios de un fichero si queremos descartar dichos cambios?*

También es muy útil la orden `git diff` que nos informa de los cambios que todavía no hemos "marcado" para incorporar al envío. Probad a modificar un fichero y ejecutad `git diff`.

Pregunta 8.11: *¿Cómo marca los cambios* `git`*?*

Si a continuación hacemos `add` del fichero con los cambios, estos desaparecen del `git diff` y aparecen entre la versión "staged" y la final y se pueden ver con `git diff --staged`

Otra opción útil es borrar ficheros de `git`, tanto del repositorio (`git rm fichero`) como del "stage" (`git reset fichero`) si decidimos que queremos olvidarnos de él. Probad a borrar alguno.

Pregunta 8.12: *¿*`Git log` *informa de los ficheros que se borran del "stage"? ¿Y de los del repositorio? ¿Y* `git status`*?*

Bueno, ya prácticamente sabemos hacer de todo en `Git`, solo nos queda poder operar con un repositorio remoto. En realidad es muy fácil. Para hacerlo vais a trabajar de nuevo en parejas de parejas. Una pareja tendrá el repositorio original (servidor) y la otra se conectará como cliente. En realidad `git` permite muy fácilmente cambiar los roles ya que todas las copias del repositorio son en realidad el propio repositorio.

Lo primero que debéis hacer es crear un repositorio para compartir (antes habíais creado un repositorio de trabajo) en un nuevo directorio vacio:

```
git init --bare
```

La diferencia básica con la vez anterior (que indica la opción `--bare`) es que ahora este directorio no está pensado para trabajar directamente en él como antes sino para ser compartido por varios usuarios (igual que lo que habíamos hecho en `CVS`). Si queréis trabajar con este repositorio desde el mismo ordenador, ahora iros a otro directorio nuevo, y clonad el directorio:

```
git clone /camino/al/repositorio/git
```

En el directorio nuevo, que será el de trabajo, ya podéis añadir ficheros con contenido, hacer el commit y enviarlos al repositorio que teníais vacio con `git push`. Aunque parece más difícil al principio ya que hemos tenido que crear dos directorios (el del repositorio y el de trabajo) ahora el repositorio se puede usar entre varios ordenadores. Para hacer esto la pareja con el ordenador servidor deberá asegurarse de tener activo el servidor `ssh`. A continuación, los clientes ejecutarán:

```
git clone usuario@pcservidor:/camino/al/repositorio/git
```

Y ya está, los clientes tendrán una copia del repositorio en local con la que podrán trabajar. Es decir, para crear un servidor `git` basta con tener acceso `ssh` a una máquina, así de simple. En realidad lo normal en proyectos de software libre es crearse una cuenta en servidores públicos tipo "GitHub" o "SourceForge", pero también podéis gestionar muy fácilmente vuestro propio servidor (acabáis de configurar uno).

Bien, ahora que sabemos trabajar con repositorios remotos, lo normal es que a veces queramos incorporar cambios sin bajar todo el repositorio que es lo que hace `git clone`. Fijaros que

como todos los repositorios son iguales lo que marca la diferencia de sentido (bajar ficheros del repositorio remoto al local o subir nuestras modificaciones) es sólo el ordenador en el que trabajamos, en realidad no hay repositorio central.

Empecemos por actualizar la información del repositorio local con lo que hay en el remoto: esto puede hacerse de dos formas: `git fetch origin` o `git pull origin`. La diferencia es que `fetch` nos bajará toda la información y a continuación nos dejará mezclar los cambios a nosotros (luego veremos algo de las ramas), mientras que `pull` mezcla directamente la rama principal remota con nuestra rama principal así que si sois novatos utilizad mejor `git pull`. `origin` es el nombre que tiene por defecto (lo pone el `clone`) el repositorio remoto a no ser que hayamos creado otro explicitamente con la orden `git remote add <nombre> <url>` (de hecho podemos tener varios repositorios remotos y trabajar con todos a la vez). La orden para subir nuestros cambios es la contraria: `git push origin master`.

Pregunta 8.13: *A continuación bajaros el repositorio principal (`git clone`) en dos directorios diferentes (uno la pareja "servidora" y otro la "cliente"), y modificad ambos el mismo fichero. ¿Os deja enviar (con la orden `git push origin master`) la modificación a ambos?*

Pregunta 8.14: *¿Cómo se puede corregir el error para poder enviar las modificaciones la segunda vez?*

Finalmente vamos a hablar un poco de las ramas. Una de las grandes virtudes de `git` es que permite muy fácilmente abrir ramas del proyecto y volver a incorporarlas al proyecto principal. Fijaros que en la última orden que hemos visto (`git push origin master`) incorporamos un nombre: `master` que ya solía aparecer en `git status` (si no fijaros ahora). Este nombre es el nombre de la rama de trabajo por defecto. El tener varias ramas sirve para tener varias versiones diferentes en las que trabajar en paralelo de forma que las más exitosas puedan incorporarse a la rama principal y el resto descartarse de forma fácil. A continuación, uno de los grupos, probad a crear una nueva rama:

```
git branch nombrerama
```

Y cambiar a ella: `git checkout nombrerama`. A continuación cambiad cosas en esa rama y que el otro grupo cambie los mismos ficheros pero en la rama principal y haced ambos el `push`.

Pregunta 8.15: *¿Con qué comando se debe hacer el `push` de la nueva rama?*

Finalmente, pasado un cierto tiempo suele haber que volver a mezclar la rama nueva (`git merge nombrerama`, la mezcla con la actual) y, seguramente eliminarla (`git branch -d nombrerama`).

Pregunta 8.16: *¿Qué comandos tiene que ejecutar el que ha creado la nueva rama para volver a mezclarla y eliminarla? ¿Hay algún problema si la nueva rama solo tiene modificaciones propias (que no comparte con la principal)? ¿Y si hay modificaciones en la misma porción del mismo archivo de las dos ramas que se quieren fusionar?*

Fijaros que siempre podéis comprobar las diferencias con la rama principal mediante el comando `git diff HEAD` (`HEAD` es un apuntador a la última versión de la rama principal, me temo que por falta de tiempo no veremos más sobre el tema).

Finalmente comentaros que hay un montón de información sobre Git en la propia página del proyecto (`http://git-scm.com/`) incluido algún libro muy bueno y algún tutorial. Es imposible ver en una clase como va todo lo que hay sobre Git pero si tenéis curiosidad os animo a echar un vistazo a las herramientas GitLab y SparkleShare.

Pregunta 8.17: *¿Para qué sirve `GitLab`? ¿Y `SparkleShare`?*