# Control de versiones con Subversion

*Ignacio Barrancos Martínez*
ignacio@adesx.com

## Contenido



# 1. Introducción al control de versiones

Cuando dos o más personas intentan trabajar con un documento compartido en red, el procedimiento normal suele ser obtener una copia local del archivo, realizar modificaciones y sustituir el archivo antiguo por el nuevo que incorpora las modificaciones. El primer problema aparece cuando alguna otra persona que obtuvo el archivo a la misma vez que nosotros, sustituye nuestra nueva versión con su versión, que obviamente no contiene nuestras modificaciones, perdiendose así todo nuestro trabajo.

La solución rápida a este problema, pasaría por usar un mecanismo de bloqueo, esto es, el usuario que desea realizar una copia del archivo, primero bloquea el archivo que desea modificar (mediante algún mecanismo del sistema operativo) y procede a realizar las modificaciones en su copia local: una vez finalizadas

las modificaciones, actualiza el documento y lo libera del bloqueo. El problema de esta solución es que sólo una persona puede trabajar sobre el documento y si por alguna razón olvida desbloquear el archivo o simplemente decide tomarse un descanso el resto de usuarios que deseen trabajar sobre el mismo archivo se verán obligados a tomar el mismo descanso.

Con el fín de evitar este problema aparecen los sistemas para el control de versiones, que permiten que varias personas obtengan una copia del documento y una vez cada uno realiza sus modificaciones, se encargan de actualizar la copia común añadiendo las modificaciones que se hayan realizado, evitando así que se pierda el trabajo, pues excepto en el caso de que se hayan realizado modificaciones incompatibles (ej. la misma parte del documento) el gestor aplicará nuestros cambios al documento almacenado en su repositorio, resultando que varias personas pueden colaborar sobre el mismo documento sin entorpecerse mutuamente.

Puede parecer que este tipo de herramientas sólo son interesantes si se trabaja en grupo pero, incluso cuando se trabaja individualmente son interesantes, pues permiten mantener una copia siempre actualizada de los documentos, evitando así el típico problema de tener varias copias del mismo documento y no poder saber cual de ellas es la 'buena'.

## 1.1. Disciplina de trabajo

Al trabajar con este tipo de herramientas colaborativas, ya sea en grupo o de forma individual, se debe seguir una disciplina de trabajo precisa, para evitar perder el trabajo realizado. Para ilustrar esta disciplina nos ayudaremos del siguiente ejemplo, donde Sally y Harry intentan trabajar sobre el mismo documento bajo un sistema de control de versiones:

1. Inicialmente en nuestro repositorio de documentos compartidos, existirá un documento A, sobre el que Harry y Sally pretenden trabajar.
2. Lo primero que ambos deberán hacer será obtener una copia local del documento, sobre el que poder trabajar: Esta operación la conoceremos como **checkout**.
3. Dependiendo del sistema de control de versiones que ambos usen, lo siguiente que alguno de ellos deberá hacer, es comunicar al otro su intención de trabajar con el documento A: Esta operación se suele llamar **bloqueo**, y no estará presente en todos los sistemas de control de versiones. *Supongamos que es Sally quien bloquea el documento A, para evitar que Harry lo modifique mientras ella está trabajando sobre él.*
4. Sally, realizará las modificaciones que desee sobre el documento, y una vez haya acabado, procederá a escribirlas en el repositorio compartido: Esta operación se conoce como **commit**.
5. Una vez el commit se haya realizado, Sally deberá **desbloquear** el fichero para permitir que Harry pueda subir las modificaciones que él hubiera realizado, siempre que el sistema implemente las operaciones de bloqueo/desbloqueo.
6. Cuando Harry intente hacer commit de los cambios que él realizó en su copia local del documento A, podrán suceder dos cosas:
   - Que Sally, aún no haya desbloqueado el documento A en el repositorio, por lo que el sistema impedirá que Harry pueda subir sus cambios, hasta que no la haya hecho Sally, y haya desbloqueado el fichero, o
   - Que Sally, haya entregado sus cambios y desbloqueado el documento A. Cuando Harry haga el commit de sus modificaciones, el sistema le avisará de que su copia local está anticuada y debe hacer un **update** de su copia local del documento A.

     Harry ejecutará esta acción, bloqueará el documento A, aplicará sus modificaciones sobre la copia actualizada que Sally modificó, y que ahora tiene en local, realizará el commit, y luego desbloqueará.

Para resumir, las acciones que principalmente formarán la disciplina de nuestro trabajo serán:

- **Checkout**: Obtener una copia a nuestro disco de los ficheros que forman parte de nuestro repositorio. El directorio local donde se copiarán estos archivos se conocerá como **cajita de arena, o sandbox** (del término en inglés).
- **Commit**: Entregar al repositorio los cambios que realizamos en los archivos de nuestra cajita

de arena.

- **Update**: Actualizar nuestra cajita de arena, con los cambios que otros usuarios realizaron sobre el repositorio.
- **Bloquear**: Marcar un archivo del repositorio, para evitar que otros usuarios suban sus cambios mientras que nosotros trabajábamos en nuestra cajita.
- **Desbloquear**: Desmarcar un archivo del repositorio, para permitir que otros usuarios puedan enviar sus cambios al repositorio.

## 1.2. Sistemas de control de versiones

Como podemos imaginar, un **sistema de control de versiones** se define como un conjunto de programas que administran el acceso a un repositorio de ficheros compartidos por varios usuarios en una red, implementan las operaciones básicas que conforman la disciplina de trabajo, y permiten consultar diferentes versiones de los ficheros que componen el repositorio: La iteracción entre los clientes (usuarios) y el sistema podrá ser: Cliente/Servidor, distribuida o P2P.

Asi, entre las distintas implementaciones de los distintos sistemas de control de versiones, encontramos ...

- Entre el **Software libre** destacamos,
  - **CVS** (http://www.cvshome.org): Es el estándard de facto, y probablemente el más conocido y extendido de entre todos los sistemas de control de versiones. Es el sistema que se usa en sitios tan conocidos como Sourceforge, Savannah, o Berlios.
  - **Subversion** (http://subversion.tigris.org): Es la alternativa más incipiente a la actual hegemonía de CVS, y se dice de él que es la evolución del CVS, prueba de ello es que Karl Fogel, autor de *Open Source Develoment with CVS* (cvshome.org), participa en el desarrollo de Subversion.
  - **Arch** (http://gnuarch.org): Otra alternativa a CVS, pero en este caso distribuida.
- Por contra, entre **Software propietario** destacan
  - **BitKeeper** (http://www.bitkeeper.com): Es la herramienta usada en desarrollo el kernel GNU/Linux.
  - **Code Co-Op** (http://www.relisoft.com/co_op/): Destaca por tratarse de un sistema distribuido P2P.
  - **Visual Source Safe** (http://msdn.microsoft.com/ssafe/): Destaca por ser el software de control de versiones de Microsoft.

# 2. Subversion

Una vez se han introducido los principios básicos de un sistema de control de versiones, se profundizará en el estudio de Subversion, como una evolución moderna del sistema de control de versiones por excelencia: CVS.

## 2.1. Instalación sobre Debian Woody

Actualmente Subversion forma parte del repositorio oficial de Debian Sarge (pruebas) y de Sid (inestable), no gozando de esta misma suerte la actual versión estable, Woody, por lo que será necesario instalarlo desde un backport: Editaremos **como usuario root** de nuestro GNU/Linux, el fichero */etc/apt/sources.lis*t y añadiremos las siguientes líneas al final del mismo:

```
-Subversion (Control de versiones)
deb http://people.debian.org/~adconrad woody subversion
deb-src http://people.debian.org/~adconrad woody subversion
```

Y luego, desde una consola de root, teclearemos ...

```
apt-get update
apt-get install subversion libapache2-svn
```

Ello nos habrá instalado, si no estaban ya , los siguientes paquetes:

- **apache2-common**: Servidor apache 2. Es necesario porque incluye soporte para Web-DAV, cosa que no estaba soportada en 1.3.29
- **apache2-mpm-worker**: Soporte mejorado para threads en Apache 2
- **db4.2-util**: Utilidades para la base de datos de Berkeley 4.2
- **libapache2-svn**: Módulo de apache para interactuar con Subversion
- **libapr0**: Librerías Apache Portable Runtime.
- **Libneon24**: Librerías para clientes Http y WebDAV.
- **Libsvn0**: Librerías que usará Subversion
- **libswig**: Librerías SWIG (Simplified Wrapper Interface Generator).
- **Subversion**: El propio sistema de control de versiones, que incluye el servidor y el cliente.

## 2.2. Creación de nuestro repositorio

Una vez se hayan descargado e instalado tanto Subversion como apache2, el siguiente paso será crear el repositorio en algún directorio de nuestro servidor.

A fín de poder ilustrar nuestros ejemplos, lo haremos sobre */var/lib/svn*, para lo que habrá que teclear como root:

```
mkdir -p /var/lib/svn
cd /var/lib/svn
svnadmin create repositorio
```

Estos comandos nos habrán creado un nuevo directorio en */var/lib/svn* llamado *repositorio*, que contendrá el conjunto de ficheros y directorios que a su vez componen nuestro primer repositorio en Subversion, formado principalmente por los archivos de base de datos de Berkeley.

Convendría que este directorio perteneciera al usuario que ejecutará apache2, presumiblemente *www-data*, además de crear un grupo en el que incluiremos a este y a todos los usuarios que deseamos puedan escribir sobre el repositorio:

```
groupadd colab
adduser www-data colab
chown -R www-data.colab /var/lib/svn/repositorio
```

## 2.3. Configuración del acceso al repositorio

Una de las principales diferencias que presenta Subversion con respecto a CVS es, que se podrá acceder a nuestro repositorio de múltiples formas, siempre que se indique su ubicación mediante una URL:

- **file://** , para acceso directo al repositorio, estando en el disco local.
- **http://** , para acceder a través de WebDAV (recomendado para extranets).
- **https://** , igual que http:// pero usando *Socket Secure Layer* (SSL).
- **svn://** , para acceso a través del protocolo svnserve (por defecto, puerto 3690), recomendado para intranets.
- **svn+ssh://** , igual que svn://, pero a través de un túnel SSH.

Dependiendo de nuestras necesidades, se configurará un método u otro, salvo para el primero de ellos,

que bastará con hacer referencia a su ubicación mediante: *file:///var/lib/svn/repositorio*.

### 2.3.1. Configuración del acceso mediante svnserve

Para configurar el acceso a través del protocolo *svnserve*, (**similar a pserver de CVS**), lo primero que debería añadir al fichero */etc/services*, son las dos líneas siguientes, si es que no estuvieran ya:

```
svn             3690/tcp    # Subversion
svn             3690/udp    # Subversion
```

Para permitir que los usuarios puedan realizar modificaciones sobre el repositorio, se deberá indicar de un modo similar a como lo hace CVS, editando los siguientes archivos del subdirectorio *conf*, donde creamos nuestro repositorio (presumiblemente en */var/lib/svn/repositorio*):

1. **Editar svnserve.conf**, dejando el siguiente contenido:

   ```
   [general]
   anon-access = read
   auth-access = write
   password-db = passwd_access
   realm = Mi Primer Repositorio
   ```

   Así se le indica al servicio svnserve, que el acceso anónimo sea de sólo lectura (*anon-access=read*), el autenticado de escritura (*auth-access=write*). Los valores permitidos serán siempre: read, write y none (si se pretende denegar el acceso).

   También se le dice a svnserve que las contraseñas deberá buscarlas en el fichero *passwd_access* (parámetro *password-db*), y el mensaje que le aparecerá al usuario para pedir su login será Mi Primer Repositorio.

2. **Crear un nuevo archivo llamado passwd_access**, con el siguiente contenido:

   ```
   [users]
   ignacio = pokemon
   rubio = pikachu
   ```

   En este archivo se especifican los login de usuarios (ignacio y rubio) y sus contraseñas (pokemon y pikachu, respectivamente). Asegúrese de que el resto de usuarios no puedan leerlo ni modificarlo,

   ```
   chmod 600 passwd_access
   chown www-data passwd_access
   ```

Para configurar la ejecución del demonio svnserve, se presentarán varias alternativas:

- **Ejecución como demonio**, que para evitar problemas con los permisos se debería lanzar como usuario *www-data*:

  ```
  su -
  su -  www-data
  /usr/bin/svnserve -d
  ```

- **Ejecución desde inet.conf**, para lo que se añadirán las siguientes líneas al fichero */etc/inetd.conf*,

  ```
  #:Subversion
  svn stream tcp nowait www-data.colab  /usr/bin/svnserve svnserve -i
  ```

  y después se reiniciará inetd, mediante ...

  ```
  /etc/init.d/inetd restart
  ```

- o bién, **ejecutarlo desde xinetd**, típico de otras distribuciones distintas a Debian, tales como Fedora: Crear un nuevo fichero en */etc/xinetd.d/svnserve*:

```
# default: on
# Subversion server

service svnserve
{
        socket_type     = stream
        protocol        = tcp
        user            = www-data
        group           = colab
        wait            = no
        disable         = no
        server          = /usr/bin/svnserve
        server_args     = -i
        port            = 3690
}
```

y luego reiniciar el servicio, mediante:

```
/etc/init.d/xinetd restart
```

Para comprobar el resultado se podrá usar el comando,

```
nmap -p 3600-3700
```

... o simplemente ...

```
netstat -utpa | grep svn
```

## 2.3.2. Configuración del acceso mediante túnel SSH

Para acceder al repositorio a través de un tunel ssh (svn+ssh), bastará con haber instalado, configurado y levantado **openssh**.

Los usuarios que se desean que puedan acceder al repositorio, deberán estar dados de alta como usuarios del sistema (*/etc/passwd*), y pertenecer al grupo al que pertenece el propietario del repositorio, que líneas antes habíamos especificado que sería el *grupo colab*:

```
adduser rubio colab
adduser ignacio colab
chown -R www-data:colab  /var/lib/svn
chmod -R g+rw /var/lib/svn/repositorio
find /var/lib/svn/repositorio -type d -exec chmod g+s {} \;
```

Además, convendría que nos asegurarámos de que todos los usuarios tiene configurado,

```
umask 002
```

añadiéndolo por ejemplo a */etc/profile*, y al script de inicio de apache2 (*/etc/init.d/apache2*).

Esto debe hacerse para que, independientemente del usuario que acceda al repositorio, cuando se creen o modifiquen archivos en el repositorio (archivos de la Base de Datos y logs principalmente), no existan problemas de permisos con los mismos: el *bit sticky* hará que permanezca el grupo propietario de los directorios, y el *umask*, que al crear nuevos archivos, estos se creen con permisos: rw-rw---.

En CVS, aunque el repositorio tenga activado el bit de sticky, no se tenía el problema de permisos de Subversion (solucionable con umask), porque este manejaba los permisos de forma independientemente sin hacer caso a umask, además de que se podía otorgar la propiedad a otro usuario unix (fichero CVS-ROOT/passwd).

Las contraseñas que se usarán para acceder al repositorio serán las mismas que estos usuarios usen para

entrar al sistema. Para, configurar este método de acceso no será necesario que esté levantado el servicio svnserve.

### 2.3.3. Configuración del acceso mediante http

Para configurar el acceso a través de http, lo primero que debería hacerse es configurar apache 2, si es que no lo estaba ya. Para aquellos usuarios acostumbrados a la primera versión de este, destacar que se encontrarán ciertas diferencias en la forma de configurarlo, aunque de una manera básica se puede hacer de la siguiente forma:

1. Cambiar el directorio de trabajo a */etc/apache2*,

```
cd  /etc/apache2
```

2. Personalizar el puerto tcp que escuchará apache, modificando el fichero *ports.conf*. Si se decide cambiar este puerto, también se deberá actualizar el fichero *sites-enabled/default*.
3. Crear al archivo con los usuarios que tendrán acceso al repositorio vía http:

```
su -
htpasswd2 -cb /var/lib/svn/svn-passwords ignacio pokemon
htpasswd2 -b /var/lib/svn/svn-passwords rubio  pikachu
```

   Con estas líneas se acaba de crear el fichero que contendrá las contraseñas (*/var/lib/svn/svn-passwords*) propiedad del root, y se han añadido dos usuarios: ignacio y rubio, con contraseñas pokemon y pikachu respectivamente.
4. Crear el archivo */var/lib/svn/svn-acl* con la lista de permisos de acceso al repositorio,

```
[/]
rubio = r
ignacio = rw
```

   Así le decimos, que el usuario ignacio podrá leer (r=read) y escribir (w=write) en él, mientras que al usuario rubio sólo se le permite la consulta (r).
5. Copiar las hojas de estilos que vienen con libapache2-svn al directorio raíz de la web:

```
cp /var/www/apache2-default/svnindex.* /var/www/
```

6. Modificar el fichero *mods-enabled/dav_svn.conf*, y dejar el siguiente contenido, personalizado de acuerdo con nuestras preferencias:

```
<Location /repositorio>
   DAV svn

   #- Ruta donde se creó el repositorio
   SVNPath  /var/lib/svn/repositorio

   #- Hoja  de estilo con la que se visualizará el contenido del repositorio.
   SVNIndexXSLT "/apache2-default/svnindex.xsl"

   #- Para proteger al acceso: Se debe usar una contraseña
   AuthType Basic
   AuthName "Repositorio Subversion"
   AuthUserFile /var/lib/svn/svn-passwords
   Require valid-user

   # Para tener una lista de control de los accesos al repositorio
   AuthzSVNAccessFile /var/lib/svn/svn-acl
</Location>
```

7. Consultar el fichero */etc/default/apache2*, y asegurarse de que su contenido es:

```
 NO_START=0
```

8. Iniciar el servicio de apache2:

```
/etc/init.d/apache2 start
```

Ya se podrá consultar en un navegador mediante *http://servidor:puerto/repositorio/*, donde *servidor* será la dirección ip del servidor donde se montó el repositorio, (127.0.0.1 si es en el propio equipo), y *:puerto*, si se modificó el puerto por defecto de apache2.

Si se quisiera personalizar el aspecto de la página web donde se muestra el contenido de nuestro repositorio, se deberá cambiar la línea *SVNIndexXSLT*, e indicar la ruta web de la nueva hoja de estilo. *Por ruta web se entiende la ruta que se teclea después de http://servidor:puerto.*

Si se pretenden simultanear las diferentes modalidades de acceso, *file:///*, *http:///*, *svn:///*, convendrá poner especial cuidado y atención en que los usuarios que accedan al repositorio tengan especificado el *umask 002*, dado que los ficheros que se podría perder el acceso, desde http, si en algún momento, el usuario que ejecuta apache2, no puediera leer o escribir algún archivo, donde creamos nuestro repositorio.

# 3. Trabajando con Subversion

Para poder trabajar con Subversion necesitará un cliente. Más adelante, se presentarán clientes gráficos, aunque de momento y para empezar, será suficiente con usar el cliente que viene con el propio servidor. Si se quiere acceder desde un equipo distinto al que instalamos el repositorio, deberemos teclear ...

```
apt-get install subversion
```

Observe cómo ello nos habrá creado el directorio */etc/subversion*. Este directorio contendrá dos archivos: *servers* y *config*, los cuales almacenan la configuración por defecto del cliente, para todos los usuarios del sistema.

Una vez, que un usuario realice cualquier operación con el cliente, se le copiarán a su directorio home, dentro de *~/.subversion*, y podrá personalizarlos según sus preferencias. A lo largo del documento se irán haciendo referencia al contenido de estos archivos, para permitirnos personalizar la configuración del cliente.

Señalar que en este directorio también se creará un subdirectorio llamado *auth*, que almacenará las URLs de los distintos repositorios a los que se acceda, junto al usuario y la clave que se usó para la autenticación (en claro), bajo el subdirectorio *svn.simple*. Con ello se evita que continuamente nos pregunte el usuario y la contraseña de acceso a cada uno de los repositorios, de forma análoga al fichero *.cvspass* del CVS.

## 3.1. Importación inicial al repositorio

Una vez se tiene un repositorio sobre el que trabajar, se necesitarán los archivos que se quieran añadir al control de versiones. Para ello se creará un nuevo directorio con un simple archivo y se realizará una importación al repositorio:

```
$ mkdir /tmp/sencillo
$ cd /tmp/sencillo
$ echo “hola mundo, de subversion” > archivo.txt
$ svn import http://servidor:puerto/repositorio/prueba  -m  "Esto es una importacion"
Adding         archivo.txt

Committed revision 1.
```

La línea que empieza por el comando *svn*, realiza la importación de todos los archivos del directorio de trabajo actual al repositorio dentro del subdirectorio llamado prueba. El parámetro *-m* especifica el men-

saje explicativo de los cambios realizados. Esta secuencia de comandos se pude usar para añadir al repositorio directorios enteros que no estaban sometidos al control de versiones.

Si en vez de realizar una importación completa, se desea simplemente crear un nuevo directorio en el repositorio, teclearemos:

```
svn mkdir http://servidor:puerto/repositorio/charla -m "Creacion de un directorio"
```

Este comando crea un nuevo directorio (*mkdir*) en el repositorio llamado charla. El parámetro *-m*, es el texto con la explicación de los cambios realizados. En los anteriores ejemplos se ha usado http para acceder al repositorio, pero siempre podríamos hacerlo mediante acceso local:

```
svn mkdir file:///var/lib/svn/repositorio/otro_directorio -m "Creacion de otro direc-
torio"
```

o bien, mediante svnserve,

```
svn mkdir svn://servidor:puerto/repositorio/otro_directorio -m "Creacion de otro di-
rectorio"
```

o incluso, las siguientes variantes, si se configurado correctamente https y/o ssh:

```
svn mkdir https://servidor:puerto/repositorio/otro_directorio -m "Creacion de otro di-
rectorio"
svn mkdir svn+ssh://servidor:puerto/repositorio/otro_directorio -m "Creacion de otro
directorio"
```

En adelante, las referencias al repositorio se realizarán mediante http dado que es una de las principales diferencias frente a cvs, aunque se deberá tener en cuenta, que cualquier otro tipo de acceso es análogo y sólo cambiará la referencia al protocolo usado en la URL.

## 3.2. Checkout: Obtener una copia en local del repositorio

Una vez se tiene acceso a un repositorio y en él, hay archivos con los que poder trabajar, lo deseable sería ver cómo podemos descargarlos a nuestro equipo para realizar modificaciones sobre ellos. Habitualmente, **esta operación se le conoce como checkout**, y consiste fundamentalmente, como se ha apuntado, en obtener una copia local del repositorio. Esta copia local se descargará sobre el directorio de trabajo donde estemos colocados en ese momento, y en el argot del control de versiones, se le suele llamar: cajita de arena (sandbox en inglés).

```
$ mkdir ~/sandbox
$ cd  ~/sandbox
$ svn checkout http://servidor:puerto/repositorio/prueba
A  prueba/archivo.txt
Checked out revision 1
```

... donde ...

- *checkout* es el comando Subversion que se quiere ejecutar: Obtener una copia local del repositorio.
- *http://servidor:puerto/repositorio/* es la URL donde está ubicado el repositorio mientras que */prueba*, indica el archivo o directorio que se quiere descargar del repositorio.
- Las siguientes líneas muestran el listado de ficheros que se han descargado, mientras la última indica el número de revisión actualmente alcanzado.

Si volvemos a ejecutar el comando, y nadie ha modificado nada del repositorio, observaremos cómo no descargará ningún archivo. **Si alguien hubiera modificado alguno de estos archivos**, al hacer de nuevo el checkout obtendríamos:

```
$ svn checkout http://servidor:puerto/repositorio/prueba
U  prueba/archivo.txt
Checked out revision 2
```

La letra *U* nos indica que se ha actualizado el fichero que teníamos en nuestra cajita de arena con la nueva versión del repositorio. Supongamos ahora, que nosotros modificamos el archivo, y alguien había modificado el archivo y subido al repositorio ...

```
$ echo “venga una prueba” >> prueba/archivo.txt
$ svn checkout http://servidor:puerto/repositorio/prueba
C  prueba/archivo.txt
Checked out revision 3.
```

Si abrimos el fichero *prueba/archivo.txt* podremos ver cómo se han perdido los cambios que nosotros habíamos realizado y en su lugar están los que realizó la última persona que entregó sus modificaciones al repositorio. La letra *C* por tanto, nos indica que el fichero ha sido cambiado, perdiendo las modificaciones que hubiéramos realizado.

En vez de usar continuamente la palabra reservada *checkout*, se puede abreviar mediante *co*, por lo que ...

```
svn checkout http://servidor:puerto/repositorio/prueba
```

sería equivalente a ..

```
svn co  http://servidor:puerto/repositorio/prueba
```

Una vez hemos jugado un poco con la casuística del checkout, observemos qué ficheros se descargan cuando realizamos esta operación:

```
$ ls -la prueba
.  ..  archivo.txt  .svn
```

Además del archivo que suponíamos debía haber (*archivo.txt*), aparece un nuevo directorio oculto llamado *.svn*. Este directorio contiene toda la información que necesita el cliente Subversion para mantenerse sincronizado con el repositorio, y se crearán con cada directorio que descarguemos. A todos estos directorios se les conoce como directorios administrativos, (en CVS estos directorios administrativos se llamaban CVS y no estaban ocultos), y conviene no borrarlos ni modificar su contenido bajo ningún concepto.

## 3.3. Commit: Actualizar el repositorio con nuestras modificaciones

Una vez hemos se ha mostrado cómo descargar la última versión de un proyecto de nuestro repositorio, convendría ver cómo almacenar los cambios realizados en el mismo, para que nuestras modificaciones estén disponibles al resto de usuarios. **Este tipo de operación se conoce como commit**, y en Subversion es análogo a como se hacía en CVS:

```
$ cd  ~/sandbox/prueba
$ echo "hola pedrito" >>  archivo.txt
$ svn commit
```

La secuencia de comandos que se ha tecleado, permite:

- Acceder al directorio local donde almacenamos la copia local del repositorio.
- Realizar una modificación cualquiera sobre *archivo.txt*,
- por último realizar el *commit* de los cambios hechos. Notar, que ahora no se necesita especificar la URL de nuestro repositorio, porque esta ya está almacenada en el directorio administrativo (*.svn*).

Al teclear este último comando, aparecerá una ventana similar a la siguiente.

```
--This line, and those below, will be ignored--
M    archivo.txt
~
~
~
"svn-commit.tmp" 4L, 67C                                       1,0-1         All
```

Esta pantalla corresponde al *editor vi*, y nos permite especificar el mensaje explicativo sobre los cambios realizados que queremos almacenar en el repositorio junto a esta versión del archivo. Se tecleará lo que se deba, y saldrá de *vi* guardando los cambios (comando ":x"), a lo que obtendremos...

```
$ svn commit
Sending        archivo.txt
Transmitting file data .
Committed revision 4.
```

Si sale de *vi* sin guardar los cambios (*:q!*), subversion entenderá que deseamos cancelar el commit actual, a lo que responderá en la consola mediante ...

```
$ svn commit

Log message unchanged or not specified
a)bort, c)ontinue, e)dit
```

Según la tecla que se pulse ...

- Si pulsa la tecla *a* y luego el *enter*, entenderá que se quiere cancelar el actual commit,
- Si pulsamos la tecla *c* y luego el *intro*, entenderá que se quiere realizar el commit, pero el mensaje explicativo estará vacío.
- Al pulsar la *e* seguida del *enter*, entenderá que queremos volver a escribir el mensaje explicativo y volverá a ponernos el editor, para darnos otra oportunidad :-)

El hecho de que sea *vi* el editor predefinido, es configurable de dos formas ...

1. Mediante la variable de entorno EDITOR, que podemos redefinir ...

   ```
   export EDITOR=/usr/bin/gedit
   ```

   para hacer que nuestro editor sea gedit. El problema de este método, es que esta variable no sólo la usa el cliente Subversion, sino muchos otros programas como CVS, SQLPLUS, etc, etc... Modificarla para Subversion, haría que se modificara para el resto de programas.
2. La otra alternativa, consiste en modificar el fichero de configuración *.subversion/config* que habrá en nuestro directorio home, y asegurarse de que se tienen las líneas ...

   ```
   [helpers]
   editor-cmd = /usr/bin/gedit
   ```

   descomentadas (sin el signo # delante).

Por último, y antes de acabar esta sección, si no se desea manejar un editor de texto para escribir los mensajes explicativos, siempre se podrán especificar en la propia línea de comandos mediante el modificador *-m* seguido del texto:

```
$ svn commit -m "Este es la explicación de todo lo que he hecho"
```

Aunque nuestro repositorio sólo tiene un archivo (*archivo.txt*), imaginemos que tenemos varios archivos y sobre ellos hemos realizado varias modificaciones. Si se ejecuta un commit como se ha visto hasta ahora, Subversion enviará todos los ficheros modificados al repositorio pero, supongamos que sólo nos inte-

resa enviar uno de los archivos modificados, y no todos: para ello se le deberá indicar el fichero a subir después de la palabra commit:

```
$ svn commit archivo.txt -m “Solución al Bug 45”
```

Si en vez de uno sólo, se quieren enviar los cambios de 2 archivos, se hará de la misma manera:

```
$ svn commit archivo.txt  otroArchivo.txt  -m “Solución a otro bug”
```

Cuando realizamos un commit, subversion calcula las diferencias entre la versión descargada y la que queremos subir mediante diffs binarios, de manera que sólo se almacenarán estas diferencias comprimidas, como archivos RCS.

Para terminar, añadir que la palabra reservada commit se puede abreviar con *ci* (*checkin*) , por lo que ...

```
svn commit archivo.txt -m “Solucion a otro bug”
```

... sería equivalente a ...

```
$ svn ci archivo.txt -m “Solución a otro bug”
```

## 3.4. Añadir nuevos archivos y directorios al repositorio

Hasta ahora se ha visto cómo con Subversion podemos ...

1. importar un nuevo directorio a nuestro proyecto o repositorio,
2. descargar archivos y directorios del repositorio, y
3. enviar las modificaciones que realicemos.

Sería interesante conocer cómo se pueden añadir nuevos archivos y directorios a un directorio que ya habíamos descargado, o lo que es lo mismo: un directorio del que se había realizado un checkout. Esto es posible gracias al comando *add* de Subversion.

Imaginemos que sobre nuestra cajita de arena, realizamos las siguientes operaciones:

1. Actualizamos nuestra copia local, como habíamos visto hasta ahora:

   ```
   $ cd ~/sandbox
   $ svn checkout http://servidor:puerto/repositorio/prueba
   ```

2. Añadimos dos ficheros de texto plano,

   ```
   $ cd ~/sandbox/prueba
   $ echo “esto es una prueba” > otroArchivo.txt
   $ echo “esto es otra prueba” > otraPrueba.txt
   ```

3. Se añade un fichero binario, como puede ser una imagen en formato jpg, gif o png ...

   ```
   $ cd ~/sandbox/prueba
   $ cp ~/fotos/sally_playa.jpg .
   ```

4. Creamos un directorio que a su vez tiene un subdirectorio, al cual añadimos otro fichero de texto plano y una imagen...

   ```
   $ cd ~/sandbox/prueba
   $ mkdir -p fotos/ejemplos
   $ echo “Estas fotos son ejemplos” > fotos/ejemplos/Leeme.txt
   $ cp ~/fotos/sally_piscina.gif fotos/ejemplos
   ```

Ahora, sólo falta indicarle a subversion que se quieren añadir todos estos archivos y directorios a nuestro repositorio. Esto se puede hacer de varias formas: O bien indicar uno a uno los ficheros y directorios que se desean añadir, o bien se le dicen todos en una misma línea, pudiendo usar comodines para ello. Elegimos esta segunda opción.

```
$ cd ~/sandbox/prueba
$ svn add  otr*txt sally_playa.jpg fotos
A            otraPrueba.txt
A            otroArchivo.txt
A  (bin)  sally_playa.jpg
A            fotos
A            fotos/ejemplos
A  (bin)  fotos/ejemplos/sally_piscina.gif
A            fotos/ejemplos/Leeme.txt
```

Observar como Subversion es capaz de detectar automáticamente qué fichero es binario, los que tienen a su izquierda (*bin*), y cuál no lo es. Además, al marcar para la adición directorios, es capaz de recorrerlos en profundidad, para añadir todos los archivos este que contenga.

Con esta acción sólo los hemos marcado para que los añada al repositorio, pero no se subirán realmente hasta que ejecutemos un commit...

```
$ svn commit -m “Prueba de la adición de ficheros”
Adding         fotos
Adding         fotos/ejemplos
Adding         fotos/ejemplos/Leeme.txt
Adding  (bin)  fotos/ejemplos/sally_piscina.gif
Adding         otraPrueba.txt
Adding         otroArchivo.txt
Adding  (bin)  sally_playa.jpg
Transmitting file data ......
Committed revision 5.
```

## 3.5. Eliminar archivos y directorios del repositorio

Una vez se ha visto cómo añadir nuevos ficheros y directorios, procedería saber cómo eliminarlos. Para ello se usará el comando *delete*, *del*, *rm* o *remove* de Subversion (todos hacen lo mismo) seguido de los archivos que se quieren eliminar:

```
$ svn del sally_playa.jpg fotos
D         sally_playa.jpg
D         fotos/ejemplos/Leeme.txt
D         fotos/ejemplos/sally_piscina.gif
D         fotos/ejemplos
D         fotos
```

Ese comando habrá marcado (*flag D*) para borrar el archivo *sally_playa.jpg* y todo el directorio *fotos*. Para que los cambios ean llevados al repositorio se deberá realizar el commit...

```
$ svn commit -m "Borrado de archivos"
Deleting       fotos
Deleting       sally_playa.jpg

Committed revision 6.
```

A diferencia de Cvs, Subversion eliminará el directorio, de manera que cuando ahora nosotros hagamos un checkout, no aparecerá un subdirectorio vacío, como sucedía con Cvs. Se puede probar:

```
$ mkdir /tmp/sandox
$ cd /tmp/sandox
$ svn checkout http://servidor:puerto/repositorio/prueba
A  prueba/pruebecita.txt
A  prueba/otraPrueba.txt
A  prueba/archivo.txt
```

```
A  prueba/otroArchivo.txt
Checked out revision 6.
```

... pero, ¿qué pasaría si quisiéramos obtener esos archivos que fueron borrados?, que simplemente con descargar una revisión en la que estuvieran presentes estos archivos (en nuestro caso la 5), sería suficiente:

```
svn checkout -r5  http://servidor:puerto/repositorio/prueba
A  prueba/sally_playa.jpg
A  prueba/fotos
A  prueba/fotos/ejemplos
A  prueba/fotos/ejemplos/Leeme.txt
A  prueba/fotos/ejemplos/sally_piscina.gif
Checked out revision 5.
```

Como se puede ver, los ficheros borrados del repositorio no se eliminan realmente: Esto es deseable que así suceda por si se desea acceder a una versión anterior de nuestro proyecto, en la que estos archivos estaban.

La novedad del comando anterior la introduce el parámetro *-r de checkout*, mediante el cuál se le indica a nuestro cliente qué versión en concreto se quiere descargar, dejando nuestra cajita de arena tal cual estaba el repositorio en ese momento.

## 3.6. Update: actualizando nuestra cajita de arena

Hasta ahora hemos visto cómo trabajar de manera elemental con un control de versiones basado en Subversion, de forma personal y unilateral: Subir nuestras modificaciones, añadir y eliminar archivos, así cómo descargar una versión del repositorio. Pero, ¿cómo se debe actualizar nuestra copia en local del repositorio, una vez se tienen varios usuarios?. Para lograrlo se debe utilizar el *comando update* de Subversion (abreviadamente up).

Con el fín de ilustrar su funcionamiento y casuística, en vez de trabajar con dos usuarios, lo haremos con dos cajitas de arena diferentes, que a todos los efectos, reflejan la problemática de tener dos o más copias del repositorio diferentes que se tendrían si hubieran varios usuarios. Para ello, imagínese la siguiente secuencia de operaciones:

1. Tenemos una copia local del repositorio en *~/sandbox1*. Se puede conseguir mediante...

   ```
   $ mkdir ~/sandbox1
   $ cd ~/sandbox1
   $ svn checkout http://servidor:puerto/repositorio/prueba
   ```

2. Y otra copia en *~/sandbox2* ...

   ```
   $ mkdir ~/sandbox2
   $ cd ~/sandbox2
   $ svn checkout http://servidor:puerto/repositorio/prueba
   ```

3. Existen dos copias de la misma versión de nuestro repositorio. Supongamos que en la primera copia modificamos el archivo *otraPrueba.txt*. Por ejemplo ...

   ```
   $ cd ~/sandbox1/prueba
   $ echo “hola esto es otra prueba más” >> otraPrueba.txt
   ```

   y sobre la segunda copia se modifica el fichero *archivo.txt*:

   ```
   $ cd ~/sandbox2/prueba
   $ echo “otra prueba más” >> archivo.txt
   ```

4. Si trasladamos esto al caso de dos usuarios, no habría ningún problema hasta ahora: Los usuarios se descargan en local una copia del repositorio, trabajan sobre ella, y luego envían sus modificaciones

mediante commit. Supongamos que uno de ellos lo hace antes que el otro (en nuestro caso, hacemos el commit de la primera de las cajitas de arena).

```
$ cd ~/sandbox1/prueba
$ svn commit -m "y venga pruebas"
Sending        otroArchivo.txt
Transmitting file data .
Committed revision 7.
```

Ahora, el segundo de los repositorios no está actualizado, o visto desde el otro punto de vista: Un usuario tiene una copia local desactualizada.

Es evidente que si este realizara un checkout perdería los cambios que había realizado sobre el *archivo.txt*, por lo que no se recomienda ejecutarlo: Para ello, lo recomendable sería ejecutar un *update*, y actualizar su copia local con los cambios realizados sobre el repositorio por el resto de los usuarios.

```
$ cd ~/sandbox2/prueba
$ svn update
U  otroArchivo.txt
Updated to revision 7.
```

Como se puede ver, el cliente ha actualizado el fichero *otroArchivo.txt* que otro usuario, o desde otra copia local del repositorio había sido modificado. Por tanto, el comando update, nos permitirá actualizar en todo momento nuestra copia local del repositorio.

Vale, pero ... ¿y si el fichero modificado hubiera sido el mismo?

```
$ cd ~/sandbox1/prueba
$ echo "vamos a ver qué pasa" >> archivo.txt
$ svn commit -m "probando,probando"
Sending        archivo.txt
Transmitting file data .
Committed revision 8.
```

¿qué pasará cuando el otro usuario actualice su copia local? ¿y si intentara hacer un commit antes de un update?

1. Intenta hacer un commit ...

   ```
   $ cd ~/sandbox2/prueba
   $ svn commit -m "entrego mis modificaciones"
   Sending        archivo.txt

   svn: Commit failed (details follow):
   svn: Your file or directory 'archivo.txt' is probably out-of-date
   svn:
   The version resource does not correspond to the resource within the transaction.
   Either the requested version resource is out of date (needs to be updated), or the
   requested version resource is newer than the transaction root (restart the com-
   mit).
   ```

   El cliente Subversion nos informa de que nuestra versión está desactualizada, y nos recomienda realizar un update antes de intentar hacer el commit.

2. Se hace pues un update,

   ```
   $ cd ~/sandbox2/prueba
   $ svn update
   C  archivo.txt
   Updated to revision 8.
   ```

... pero... ¿qué ha pasado con los cambios que realizamos antes de hacer el update?. La mejor forma de averiguarlo es editarlo el fichero. Observar cómo aparecerán unas líneas como las siguientes:

```
<<<<<<< .mine
```

```
otra prueba más
=======
vamos a ver qué pasa
>>>>>>> .r8
```

dónde se marca qué teníamos nosotros (*.mine*) y qué hay nuevo en la versión 8 (*.r8*). Si listamos el contenido del directorio actual veremos cómo además de *archivo.txt* aparecen los siguientes archivos:

- *archivo.txt* : El archivo que está pendiente de subir, y que otro usuario modificó mientras nosotros realizábamos nuestros cambios, con las marcas de qué es lo que ha hecho cada uno.
- *archivo.txt.mine*: Es el archivo que teníamos con nuestros cambios, justo antes de hacer el update
- *archivo.txt.r7*: Archivo original sobre el que nosotros empezamos a escribir nuestras modificaciones
- *archivo.txt.r8*: Archivo actualizado de acuerdo con la última revisión que hay de él en el repositorio.

Este conflicto se produce porque ambas versiones editaban la misma línea del fichero (la última), y entonces Subversion prefiere delegar la responsabilidad de decidirse por unos cambios u otros al propio usuario.

Para solucionar el conflicto, se editará el fichero archivo.txt, dejando los cambios que consideremos oportunos, y borrando las líneas extras que metió el cliente. Luego se ejecutará ...

```
$ svn resolved archivo.txt
Resolved conflicted state of 'archivo.txt'
```

... lo que nos borrará los archivos auxiliares que creó y para consolidar los cambios sobre nuestro repositorio, realizaremos el commit ...

```
$ svn commit -m “entrego mis cambios, una vez resuelto el conflicto"
Sending        archivo.txt
Transmitting file data .
Committed revision 9.
```

Es interesante observar cómo si en cada uno de los repositorios se hubiera editado el mismo archivo, pero distintas líneas, el propio cliente al realizar el update fusiona las dos versiones sin crear estos archivos intermedios y sin necesidad de ejecutar el *comando resolved*. Se deja como ejercicio su demostración, a la que debe obternerse cuando se realice el update algo similar a lo siguiente (*Flag G*, que indica que ha realizado la fusión):

```
$ svn update
G  archivo.txt
Updated to revision 10.
```

Este comportamiento de subversion también es aplicable en la misma medida a ficheros binarios, lo cual hace que se pueda **prescindir del comando lock y unlock que teníamos en Cvs**, aunque con este último también teníamos el comando merge que presentaba un comportamiento similar.

## 3.7. Conocer el estado de nuestra cajita de arena

La sección anterior nos ha permitido introducirnos en la problemática que existe al acceder concurrentemente a los mismos archivos en un repositorio compartido. Ello crea la necesidad de disponer de comandos que permitan conocer en todo momento, el estado en el que se encuentra nuestra copia local del repositorio con respecto al repositorio central. Subversion dispone de cinco comandos para ello: *status*, *info*, *log*, *diff* y *revert* que veremos a continuación.

El primero de ellos nos permitirá saber el estado en el qué están los archivos de nuestra copia local. Para ilustrar su funcionamiento seguiremos con trabajando con dos copias en local como en el apartado anterior.

1. Lo primero que debería hacerse es actualizar las dos copias locales:

```
$ cd ~/sandbox1/prueba  &&  svn update
$ cd ~/sandbox2/prueba  &&  svn update
```

2. Modificamos en la primera cajita *archivo.txt*, creamos dos archivos, pero sólo marcamos uno de ellos para subir, y borramos el fichero *otroArchivo.txt*:

```
$ cd ~/sandbox1/prueba
$ echo "prueba, del estado" >> archivo.txt
$ echo "este para añadir" > nuevo1.txt
$ echo "este no se añadirá " > nuevo2.txt
$ svn add nuevo1.txt
A         nuevo1.txt
$ svn del otroArchivo.txt
D         otroArchivo.txt
```

3. En la segunda cajita de arena, modificamos el fichero *otraPrueba.txt* y lo subimos al repositorio:

```
$ cd ~/sandbox2/prueba
$ echo "vamos a ver lo que pasa" >> otraPrueba.txt
$ svn commit -m "Vamos, arriba"
Sending        otraPrueba.txt
Transmitting file data .
Committed revision 11.
```

Ahora podríamos comprobar en qué estado está la primera cajita de arena con respecto al repositorio, mediante **el comando *status***: para ello,

```
$ cd ~/sandbox1/prueba
$ svn status
?       nuevo2.txt
M     archivo.txt
A      nuevo1.txt
D      otroArchivo.txt
```

El cliente nos muestra un informe de lo que haría si en este momento hiciéramos un commit: Entregar las modificaciones de *archivo.txt*, añadir *nuevo1.txt*, borrar *otroArchivo.txt*, y no sabría qué hacer con *nuevo2.txt*, que de repente ha aparecido en la cajita de arena, y no se le ha dicho qué debe hacer con él.

En general, el significado del conjunto de flags que siempre encontraremos, en cualquier operación svn es:

1. A : El archivo o directorio ha sido marcado para ser añadido al repositorio.
2. C : El fichero tiene un conflicto.
3. D : El fichero o directorio ha sido marcado para ser eliminado.
4. G : El fichero ha sido fusionado con la copia del repositorio. Esto sucede como vimos en el apartado anterior, cuando modificamos un archivo en local, y la copia del repositorio también fue modificada: Si cambiaron distintas líneas del archivo, Subversion fusionará los cambios en uno fichero y lo marcará con G. En otro caso aparecerá un conflicto.
5. M : El contenido del archivo ha sido modificado desde la última vez que se descargó.
6. U : El fichero local fue actualizado con la versión del repositorio.
7. X : El directorio no está bajo el control de versiones, pero probablemente pertenezca a una rama anterior.
8. ? : El fichero o directorio no está bajo el control de versiones.
9. ! : El fichero o directorio está bajo el control de versiones, pero no parece estar completo. La causa podría ser que se cancelara un update o un checkout, o se borrara a mano sin usar svn delete
10. ~ : El archivo está bajo el control de versiones, pero no coincide el tipo de objeto: Esto es habitual cuando se borra a mano un fichero y luego se crea un directorio con ese nombre, antes de haber hecho el commit.

Sin embargo, *svn status* (que puede abreviarse mediante *stat* o *st*), no ha hecho ninguna mención a que el fichero *otraPrueba.txt* ha sido modificado: Ello es, porque para averiguar el estado de nuestra copia local, se basa en el contenido de los subdirectorios administrativos (*.svn*), que fueron actualizados la última vez que se accedió al repositorio.

En consultas sobre el estado de nuestra copia local, también resulta útil **el comando *info***, que se puede usar sin argumentos, (mostrará información sobre el directorio de trabajo), o especificando distintos nombres de archivos o directorios, con lo que para cada uno mostrará su correspondiente información:

```
$ cd ~/sandbox1/prueba
$ svn info otroArchivo.txt
Path: otroArchivo.txt
Name: otroArchivo.txt
URL: http://helicon/repositorio/prueba/otroArchivo.txt
Repository UUID: c3bbae41-07d6-0310-8343-d679cbdbf2ac
Revision: 10
Node Kind: file
Schedule: delete
Last Changed Author: ignacio
Last Changed Rev: 7
Last Changed Date: 2004-08-09 23:50:39 +0200 (Mon, 09 Aug 2004)
Text Last Updated: 2004-08-09 23:47:34 +0200 (Mon, 09 Aug 2004)
Properties Last Updated: 2004-08-09 23:36:16 +0200 (Mon, 09 Aug 2004)
Checksum: fccfb62faf34d92e548f56b1ca7daf94
```

Como se podrá apreciar el comando arroja un montón de información, donde destacan la URL del archivo, la revisión del directorio actual, la acción que se ejecutará cuando se ejecute el commit (*Schedule*), la última revisión en la que se modificó el archivo, y su autor.

Al igual que sucediera con status, este comando tampoco accede al repositorio, sino que se basa en la información que hubiera en el directorio administrativo (.svn). Si lo que se desea es conocer el histórico de cambios que se han realizado sobre un archivo o directorio (sin argumentos) determinado, se puede usar **el comando *log***:

```
$ svn log otroArchivo.txt
------------------------------------------------------------------------
r7 | ignacio | 2004-08-09 23:50:39 +0200 (Mon, 09 Aug 2004) | 1 line

y venga pruebas
------------------------------------------------------------------------
r5 | ignacio | 2004-07-25 21:56:52 +0200 (Sun, 25 Jul 2004) | 1 line

Prueba para añadir ficheros
------------------------------------------------------------------------
```

Esto, habitualmente se conoce como el changelog de un archivo o directorio, y básicamente mostrará la revisión en la que se produjo algún cambio, su autor, la hora, y el mensaje que escribió al realizar el commit. Su funcionamiento es similar al de Cvs, y tampoco accede al repositorio, sino que trabaja con el contenido de los directorios administrativos.

También podemos usar el modificador *–xml*, para obtener el resultado de la consulta en XML:

```
$ svn log otroArchivo.txt –xml
<?xml version="1.0" encoding="utf-8"?>
<log>
<logentry
   revision="64">
<author>ignacio</author>
<date>2004-08-09T21:50:39.061195Z</date>
<msg>y venga pruebas</msg>
</logentry>
<logentry
   revision="61">
<author>ignacio</author>
<date>2004-07-25T19:56:52.102043Z</date>
<msg>Prueba para aÃ±adir ficheros</msg>
</logentry>
</log>
```

Si se quisieran conocer los cambios que se realizaron en cada una de las revisiones se podrá usar el **comando *diff*** o abreviadamente *di*.

```
$ svn diff -r 7:5 otroArchivo.txt
Index: otroArchivo.txt
===================================
--- otroArchivo.txt     (revision 7)
+++ otroArchivo.txt     (revision 5)
@@ -1,2 +1 @@
 esto es una prueba
-hola esto es otra prueba más
```

Como argumentos le hemos pasado los números de revisión que se querían examinar (-r) y el archivo que se queria visualizar. En la salida del comando, Subversion presenta una pequeña leyenda que indicará las líneas que vayan precedidas por + y - a qué revisión corresponden. Luego irá mostrando los bloques de diferencias precedidos por @@, donde...

- La primera tupla (después de @@ y antes del espacio) precedida del -, indica el intervalo de líneas (separado por coma) que se están mostrando para la revisión etiquetada con – (en nuestro ejemplo la 7): *En el ejemplo anterior se indica desde la uno a la dos.*
- La segunda tupla (después del espacio hasta los siguientes @@) precedida por el signo +, indica el intervalo de líneas mostrado para la revisión etiquetada con +++ (la 5 en nuestro ejemplo). *En el caso presentado, el intervalo sólo está acotado por un número (la línea uno), porque el archivo para esa revisión sólo contenía una línea.*

Lo siguiente serán las líneas que componen los diferentes intervalos, donde si no van precedidas por ningún signo, indica que se comparten en ambas revisiones, y si van precedidas por alguno de los dos símbolos (+ o -), hacen referencia al número de revisión en la aparecen.

Este comando nos resulta de suma utilidad cuando se quieren preparar parches de actualización:

```
$ svn diff -r 5:7  otroArchivo.txt  >  update-to-rev7.patch
```

Este comando permitirá recoger en el fichero update-to-rev7.patch, el parche para subir de la versión 5 a la 7. Para probarlo, ejecutar ...

```
$ svn update -r 5 otroArchivo.txt
At revision 5.

$ patch otroArchivo.txt  update-to-rev7.patch
```

El siguiente comando que veremos es análogo al *Undo* o *Deshacer* de cualquier editor o programa de edición: ***svn revert*, permite deshacer todos los cambios que hubiéramos realizado sobre uno o más archivos de nuestra copia local**, retornando a la versión que descargamos la última vez. Para problarlo...

```
$ cd ~/sandbox1/prueba
$ svn revert otroArchivo.txt
Reverted 'otroArchivo.txt'
```

A lo que nos informa, que ha “revertido” nuestro fichero (ha deshecho los cambios). Si ahora ejecutamos ...

```
$ svn status
?       nuevo2.txt
M      archivo.txt
A       nuevo1.txt
```

Podemos observar como ya no indica que se ha marcado el archivo *otroArchivo.txt* para ser borrado, y si hiciésemos ahora commit, sólo se entregarían las modificaciones de *archivo.txt*, y se añadiría *nuevo1.txt* al repositorio.

Al igual que sucediera con los otros comandos, revert no necesita acceder al repositorio, sino que se basa en el contenido de los directorios administrativos. Por el contrario, requiere que se le indiquen como argumentos los ficheros sobre los que se desean deshacer los cambios.

## 3.8. Manipular los archivos del repositorio

Desde las primeras versiones de Subversion, se hace latente el deseo de los desarrolladores en mejorar ciertos aspectos negativos que presentaba Cvs. Cvs tenía grandes carencias en cuanto a la manipulación de los archivos del repositorio, teniendo que ir la mayoría de las veces un administrador con permisos de escritura sobre nuestro repositorio, a realizar los pertinentes cambios in situ, con el conseguiente riesgo de causar daños irreparables sobre el control de versiones de nuestros proyectos.

Bien por este deseo, bien porque el repositorio es una base de datos en la que un adminsitrador no puede meter las manos para modificar a su antojo, lo cierto y verdad, es que Subversion incorpora una serie de comandos que permiten interactuar con los ficheros del repositorio, de la misma manera que lo haríamos desde una consola o desde un explorador de archivos.

El primero de ellos y por analogía con Cvs es el comando *mkdir*. Este comando nos permite **crear un directorio en nuestro repositorio sin necesidad de crearlo desde el sistema operativo** y luego añadirlo con *svn add*.

```
$ svn mkdir unDirectorio
A         unDirectorio
```

Observar como Subversion se encarga de crearlo en el sistema de archivos. Obviamente, el directorio no se creará en el repositorio hasta que no hagamos el commit. Si el directorio ya hubiera sido creado desde el sistema operativo, subversion nos advertiría de que debemos usar *svn add*:

```
$ mkdir otroDirectorio
$ svn mkdir otroDirectorio
svn: Try 'svn add' or 'svn add --non-recursive' instead?
svn: Can't create directory 'otroDirectorio': File exists
```

Otro comando que se hacía necesario en Cvs es el de copia: *svn copy* o abreviadamente *svn cp*, **nos permite sacar copias de los archivos de nuestro repositorio manteniendo la historia**.

```
$ svn copy  otroArchivo.txt unaReplica.txt
A         unaReplica.txt
$ svn commit -m "prueba para copiar archivos"
Adding         unaReplica.txt

Committed revision 15.
```

Si ahora ejecutamos ...

```
$ svn log unaReplica.txt
------------------------------------------------------------------------
r15 | ignacio | 2004-08-17 00:16:26 +0200 (Tue, 17 Aug 2004) | 1 line
prueba para copiar archivos
------------------------------------------------------------------------
r7 | ignacio | 2004-08-09 23:50:39 +0200 (Mon, 09 Aug 2004) | 1 line
y venga pruebas
------------------------------------------------------------------------
r5 | ignacio | 2004-07-25 21:56:52 +0200 (Sun, 25 Jul 2004) | 1 line
Prueba para añadir ficheros
------------------------------------------------------------------------
```

A partir de este momento el histórico de cambios se mantendrá por separado.

Y como no podía ser de otra forma, **Subversion cubre la necesidad de renombrar y mover archivos manteniendo su historia**, mediante el comando *svn move*, también de forma abreviada: *mv*, *rename o*

.

```
$ svn move unaReplica.txt  replica.txt
A         replica.txt
D         unaReplica.txt
```

Como se puede ver, internamente hace una copia y borra el original. Como la copia nos conserva la historia, el renombrado también. Se puede probar *svn log replica.txt* tras el commit.

Cuando se trabaja a diario con un control de versiones, es útil **conocer el contenido de un determinado directorio para una versión en concreto, sin tener que bajarla por completo**. Subversion soluciona esta necesidad mediante *svn list* (abreviadamente ls), al que podremos indicarle el número de revisión concreto que queremos listar. Si se omite, entenderá que se quiere conocer el listado actual de la última revisión, que no tiene por qué coincidir para nada con nuestra copia local. A diferencia de otros comandos que se servían del contenido de los directorios .svn, este comando sí necesitará poder acceder al repositorio para ejecutarse.

```
$ svn list -r 1
archivo.txt

$ svn list -r 9
archivo.txt
otraPrueba.txt
otroArchivo.txt
pruebecita.txt
```

De forma análoga al comando list, Subversion dispone el comando *cat*, que **volcará el contenido de un archivo, o una revisión concreta de él, accediendo al repositorio** y sin la necesidad de tener que hacer el *svn update*, y modificar con ello nuestra copia local:

```
$  svn cat -r6 otroArchivo.txt
esto es una prueba
```

Observar como si intetamos mostrar el contenido de una revisión anterior a la copia de un archivo, Subversion nos dará un error incándonos que no se encuentra:

```
$ svn cat -r14 replica.txt
svn: PROPFIND request failed on '/repositorio/!svn/bc/14/prueba/replica.txt'
svn: '/repositorio/!svn/bc/14/prueba/replica.txt' path not found
```

## 3.9. Trabajar con la metainformación que almacena Subversion

En CVS era habitual y muy útil, trabajar con autotags como *Id*, *Author*, *Date*, etc... Esto eran tags especiales, que CVS era capaz de expandir automáticamente cuando nosotros hacíamos un commit, así, en el propio código fuente quedaba un registro del autor del fichero fuente, la fecha de la última modificación, etc, etc...

Uno de los inconvenientes de usar Subversion es que esta funcionalidad no está habilitada por defecto, y puede suponer una molestia para aquellos usuarios que venimos de Cvs. En esta sección se verá cómo habilitar esta característica para conseguir que Subversion se comporte como su predecesor, yendo incluso más allá.

Para ello, lo primero será saber qué autotags podemos establecer para que sean expandidos por el cliente: Ejecute *svn help ps*, y observe el apartado dedicado a *svn:keywords*. En él se enumeran todos los posibles autotags soportados por el cliente.

Para empezar a experimentar con ellos, crearemos un archivo simple en nuestro repositorio, llamado *propiedades.txt*, con el siguiente contenido:

```
$ cat ~/sandbox1/propiedades.txt <<EOF
```

```
Vamos a ver las propiedades definidas por
defecto para svn:keywords
----8<---------------------------------------
$URL$
$HeadURL$
$Author$
$LastChangedBy$
$Date$
$Rev: 107 $
$LastChangedRevision: 107 $
$Id: subversion_vs_cvs.html.xml 107 2005-03-21 18:48:29Z ignacio $
----8<---------------------------------------
Aquí terminan
EOF
```

Si ahora entregáramos el fichero al repositorio, no pasaría absolutamente nada. Antes habrá que decirle al cliente que queremos habilitar esta funcionalidad:

```
$ svn add propiedades.txt
$ svn propset svn:keywords “URL HeadURL Author LastChangedBy Date Rev LastChangedRevi-
sion Id” propiedades.txt
$ svn commit -m “Prueba de propset”
```

Si ahora editamos el contenido del fichero, observaremos cómo se han expandido todas estas propiedades:

```
Vamos a ver las propiedades definidas por
defecto para svn:keywords
----8<---------------------------------------
$URL: http://helicon/repositorio/prueba/propiedades.txt $
$HeadURL: http://helicon/repositorio/prueba/propiedades.txt $
$Author: ignacio $
$LastChangedBy: ignacio $
$Date: 2004-08-29 12:56:19 +0200 (Sun, 29 Aug 2004) $
$Rev: 107 $
$LastChangedRevision: 107 $
$Id: subversion_vs_cvs.html.xml 107 2005-03-21 18:48:29Z ignacio $
----8<---------------------------------------
Aquí terminan
```

En consecuencia, podríamos concluir que el significado de cada uno de estos tags es el siguiente:

- **URL, HeadURL**: URL del fichero en el repositorio.
- **Author, LastChangedBy**: Usuario que realizó la última modificación
- **Rev, LastChangedRevision**: Revisión en la que se realizó la última modificación.
- **Id**: Resúmen de las otras propiedades.

Lo interesante de este comportamiento, es que esta información se manipula como metainformación (información sobre la información), por lo que si modificamos el archivo, entregamos al repositorio y luego aplicamos *svn diff* entre las dos versiones, veremos cómo sólo encontrará diferencias entre las líneas modificadas, no viéndose afectada las líneas que contienen los autotags, al contrario que si usarámos el comando *diff* del sistema operativo.

Para ser sinceros, el que continuamente tengamos que decirle al cliente en qué archivos debe usar esta funcionalidad, resulta molesto y “cansino”, por lo que puede ser interesante configurar el cliente para que la habilite a todos los ficheros. Para lograrlo, edite el fichero *~/.subversion/config*, y al final de la sección *[miscellany]* añada:

```
enable-auto-props = yes

[auto-props]
* = svn:keywords=Id Rev
```

Podrá observar cómo si ahora añade un nuevo fichero al repositorio y en él pone alguno de estos autotags,

automáticamente Subversion los expandirá. Destacar que la expansión de estas propiedades se realiza al hacer un *update* o un *commit*: Si intentamos acceder al repositorio vía web (http/https) y visualizar alguno de estos ficheros, observará como estas propiedades no aparecen expandidas, sino comprimidas. Si se quisieran añadir más propiedades, tan sólo tendríamos que añadirlas al final de la línea separadas por un espacio.

Esto presenta un gran inconveniente, y es qué pasa con los ficheros binarios que tengan una cadena *$Id: subversion_vs_cvs.html.xml 107 2005-03-21 18:48:29Z ignacio $* ó *Rev*. Pues sencillamente, que expandirá estas propiedades. Podría comprobarlo creando un simple fichero en C (llamado *binario.c*, por ejemplo), con el siguiente contenido:

```
#include <stdio.h>
/*
 $Id: subversion_vs_cvs.html.xml 107 2005-03-21 18:48:29Z ignacio $
 */
int main() {
  printf("Hola esto una prueba\n");
  printf("$Id: subversion_vs_cvs.html.xml 107 2005-03-21 18:48:29Z ignacio $\n");
}
```

Luego compílelo mediante ...

```
gcc -o binario binario.c
```

Compruebe el resultado de ejecutar el fichero binario ...

```
$ ./binario
Hola esto una prueba
$Id: subversion_vs_cvs.html.xml 107 2005-03-21 18:48:29Z ignacio $
```

Y añada los dos archivos al repositorio:

```
$ svn add binario*
$ svn commit -m “prueba con binarios”
```

Verifique que se han expandido las propiedades en los dos ficheros , y pruebe de nuevo a ejecutarlo: Le dará un error. Para evitarlo, tenemos dos posibilidades:

1. Indicar en el fichero de configuración (*~/.subversion/config*), una a una todas las extensiones de los archivos que queremos expandir estas propiedades,

   ```
   *.c = svn:keywords=Id Rev
   *.cpp = svn:keywords=Id Rev
   Makefile = svn:keywords=Id Rev
   *.xml =  svn:keywords=Id Rev
   ```

   ...etc...
2. O bien usar el comando de subversion que elimina esta funcionalidad para archivos concretos, una vez lo hayamos creado. En el ejemplo anterior, bastaría con haber tecleado ...

   ```
   svn propdel svn:keywords binario
   ```

   antes de ejecutar el commit.

Existen otras propiedades muy importantes además de las que Subversion comparte con CVS:

- ***svn:eol-style***: Indican cómo debe procesar los retornos de carro el cliente Subversion. Es antológico el problema que existe entre los retornos de carro de Windows y Unix, y se ve claramente cuando editamos un fichero Unix con Notepad en Windows, el cual mete cuadritos donde debiera haber un retorno de carro, y todo el contenido seguido.

Los clientes CVS (por lo menos los que yo he manejado), saben cuando se encuentran en una plataforma Windows y cuando en una plataforma Unix, y realizan las sustituciones pertinentes de forma automática. En Subversion se debería indicar de manera explícita, así, lo más razonable sería añadir líneas a nuestro fichero *~/.subversion/config*, dentro de la sección *[auto-props]*, como se muestra a continuación:

```
*.c = svn:eol-style=native
*.cpp = svn:eol-style=native
*.h = svn:eol-style=native
*.bat = svn:eol-style=CRLF
*.sh = svn:eol-style=LF
```

De esta forma, Subversion entiende que los ficheros con extensión c,cpp y h, los retornos de carro se transformarán según marque el sistema operativo (automáticamente detectará cuando está en Windows y cuando sobre Unix), mientras que para los archivos .bat entenderá que siempre debe transformarlos al formato Windows. Los archivos .sh se les fuerza a que tenga el retorno de carro de Unix. Para probarlo deberíamos crear ficheros con Windows y Linux, y hacer checkouts, modificaciones y commits desde ambas plataformas.

- ***svn:executable***: Le dice al cliente Subversion que estos archivos debe activarles el bit de ejecutables cuando haga el checkout o el update. De la misma forma que antes, editaremos el fichero *~/.subversion/config*, y dentro de la sección *[auto-props]*, añadir ...

```
*.sh = svn:executable
```

o si se quiere habilitar de forma individual ...

```
svn propset svn:executable fichero
```

- ***svn:mime-type***: Si exploramos nuestro repositorio con el navegador web, observaremos cómo al hacer click sobre los ficheros de texto plano se nos muestra sin más en la ventana del navegador.

  Si hacemos click sobre un fichero de imagen, como puede ser un png, jpg o gif, el navegador nos preguntará que qué queremos hacer con este archivo [application/octect-stream]. Esto es causado porque Subversion almacenó que el tipo mime para ese archivo era binario sin más.

  Para modificar el tipo mime de un fichero usaremos la propiedad *svn:mime-type*, de esta forma ...

```
svn propset svn:mime-type image/png archivo.png
```

o a través del fichero de configuración, añadiendo la línea ...

```
*.png = svn:mime-type=image/png
```

En el fichero de configuración podemos indicar varias propiedades para la misma extensión en una sóla línea, separándolas mediante el caracter punto y coma:

```
*.sh = svn:eol-style=native;svn:executable
```

Subversion también permite definir nuestras propias propiedades mediante propset. Imaginemos que se quiere añadir como metainformación de un fichero, el Copyright. Procederíamos mediante:

```
svn propset copyright '(c) Ignacio' fichero.
```

Esto no hará que automáticamente se expanda cada ocurrencia de la palabra copyright en fichero, por el texto *(c) Ignacio*. Esto sólo sucede con las propiedades *svn:keywords* conocidas (las que vimos líneas más arriba).

Para extraer la metainformación asociada a este archivo, usaremos ...

```
svn propget copyright fichero
```

# 4. El día a día con Subversion

En esta sección he querido recoger mi experiencia en el día a día con Subversion, mostrando aquellas cosas que trascienden durante el trabajo diario.

## 4.1. Pequeñas cosas y truqillos

Aquí se ampliarán esas pequeñas cosas que sabíamos hacer con CVS, y que ahora cambian con respecto a Subversion.

### 4.1.1. Fechas de los archivos tras un checkout

Uno de los primeros inconvenientes que encontramos los usuarios de CVS, que empezamos con Subversion, es que cada vez que realizamos un checkout, el cliente nos fija las fechas de los archivos, a la fecha en la que realizamos el checkout.

Esto resulta muy incómodo si en nuestro repositorio tenemos Makefiles, que permiten construir nuestro proyecto. Para poder establecer, que al hacer checkout, se fijen las fechas de la última vez que se modificó el archivo en el repositorio, deberemos editar el fichero *~/.subversion/config*, y añadir dentro de la sección *[miscellany]*:

```
use-commit-times = yes
```

### 4.1.2. Ignorar archivos

Al trabajar con los ficheros de nuestro repositorio, es habitual que los diferentes editores con los que los manipulamos, creen archivos temporales, que al hacer el commit, intentan ser enviados al repositorio.

Para evitar que continuamente nos aparezcan al interactuar con el cliente de subversion, podemos definirlos en el fichero *~/.subversion/config*, dentro de la sección *[miscellany]*, mediante ...

```
global-ignores = *.o *.lo *.la #*# .*.orig *.orig .*.rej *.rej .*~ *~ .#* *.bak
*.class .*.swp
```

### 4.1.3. Problemas con las tildes

Al escribir los mensajes de log cuando se hace commit, se puede encontrar con errores indicándole que determinados caracteres no son válidos. Ello se debe a que el charset que espera el servidor, no concuerda con el que le envía el cliente, y este último, se debe asegurar de transformar los mensajes a un charset que acepte el servidor:

Ello se define en el fichero *~/.subversion/config*, dentro de la sección *[miscellany]*, mediante ...

```
log-encoding = latin1
```

### 4.1.4. Cambio de subred manteniendo nuestra copia local

Cunado trabajamos, no siempre lo hacemos en la misma ubicación o subred, y es habitual que tengamos movilidad, de forma que la URL para acceder al repositorio cambiará en función del lugar donde nos encontremos:

Desde casa, podremos acceder a nuestro repositorio mediante *http://192.168.1.20*, mientras que desde el

trabajo, lo haremos mediante *http://micasa.homelinux.org*.

Esto nos generará un problema a la hora de mantener nuestra copia local del repositorio, dado que en los directorios administrativos (.svn), está escrita la URL desde donde hicimos el último checkout. Por suerte, el cliente de Subversion añade una funcionalidad extra que nos permite renombrar estos archivos administrativos, y poder trabajar con normalidad sobre el repositorio:

```
svn switch --relocate  http://oldhostname   http://newhostname
```

## 4.2. Administración de Subversion

Bajo este título he querido recoger los aspectos más fundamentales y triviales para administrar un repositorio de Subversion, y sobre todo, los errores que nos aparecerán, a poco que trabajemos con él.

El principal foco de problemas, suele aparecer con la compartición de diferentes métodos de acceso al repositorio: Es aconsejable usar uno de ellos, o http/https, o svn, o file/svn+ssh, pero no todos, indistintamente. Poderse se puede, pero a poco empecemos surgirán las primeras corrupciones de la Base de datos.

### 4.2.1. Error: Could not open the requested SVN filesystem

El típico error que aparece cuando se corrompe la base de datos. Desde el navegador, se nos mostrará un XML como el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<D:error xmlns:D="DAV:" xmlns:m="http://apache.org/dav/xmlns" xmlns:C="svn:">
<C:error/>
<m:human-readable errcode="160029">
Could not open the requested SVN filesystem
</m:human-readable>
</D:error>
```

Y desde el cliente ...

```
svn: Berkeley DB error while opening environment for filesystem /
var/lib/svn/repositorio/db:
DB_RUNRECOVERY: Fatal error, run database recovery
```

Para solucionarlo, se debe ser root y teclear desde una consola ...

```
# /etc/init.d/apache2 stop
# svnadmin recover /var/lib/svn/repositorio/
# chown -R www-data:colab  /var/lib/svn/repositorio/db
# chmod g+w /var/lib/svn/repositorio/db
# /etc/init.d/apache2 start
```

### 4.2.2. Copia de seguridad de nuestro repositorio

Una vez se acostumbra uno a usar un control de versiones, todo su trabajo se almacena en él, siendo de vital importancia tener copias de seguridad actualizadas del mismo:

- **Para crear una copia de seguridad de nuestro repositorio**:

```
fecha=$(date "+%Y%m%d")
svnadmin dump file:///var/lib/svn/repositorio | gzip -f > reposito-
rio.dump-$fecha.gz
```

- **Para recuperar nuestro repositorio, desde una copia de seguridad** ...

```
svnadmin load file:///var/lib/svn/repositorio < backup.dump
```

## 4.3. Clientes gráficos

Aunque en todo el documento hayamos estado trabajando con el cliente para línea de comandos, que tenemos disponible en Debian, ello no quita que no existan clientes gráficos, que a fín de cuentas, nos evitarán tener que aprendernos todos los comandos desde la consola. Son los siguientes:

- eSvn: Parece orientado a KDE, y no lo he probado.
- RapidSVN : Basado en GTK, es el cliente que actualmente uso en Linux. En cuanto a funcionalidad y configurabilidad es similar a WinCVS como cliente CVS en Windows.
- jsvn : Aplicación Java, que anteriormente usaba: Es un poco cutre, aunque funciona bien, y tiene la ventaja de ser multiplataforma.
- tortoiseSVN: Una pena que sólo esté disponible para Windows. Este el mejor cliente de los que he tenido ocasión de probar: Se integra con el Explorer de Windows, y al hacer click con el botón derecho del ratón, aparece un menú contextual con las opciones que puedes realizar: Es impresionante.

## 4.4. Para saber más

Aunque este documento no sea definitivo, y mi intención es ir enriqueciéndolo con mism propias aportaciones, según vayan surgiendo, a continuación expongo unos links que ayudarán a ampliar el conocimiento sobre Subversion:

- http://www.cs.put.poznan.pl/csobaniec/Papers/svn-refcard.pdf
  Excelente guía de referencia rápida, donde en un folio por ambas caras nos resume todos posibles comandos que tenemos disponibles.
- http://svnbook.red-bean.com/
  Libro oficial de subversion en inglés.
- http://bulma.net/body.phtml?nIdNoticia=1985
  Artículo de bulma, Revoluciona tu código con Subversion, que viene a ser una pequeña introducción al mismo.
- http://libertonia.escomposlinux.org/story/2004/5/19/142850/344
  Subversion y Gentoo.

# 5. Comparativa con CVS

La primera vez que instalé Fedora Core 2, me llamó mucho la atención uno de los anuncios informativos que aparecen mientras esperamos que se instalen el sistema en nuestro disco duro, en el cual se me anunciaba que se instalaría Subversion, y textualmente el mensaje decía "Un herramienta como CVS pero mejor": Me dejó perplejo.

En mi modesta opinión y en base a mi experiencia con uno y otro, esto es poco menos que discutible. A mi juicio, las principales diferencias entre uno y otro, que yo destacaría, son las siguientes:

1. **Commits atómicos en subversion**: En Subversion, cuando hacemos un commit de un grupo de archivos, o se hacen todos, o no se hace ninguno. Esto que parece elemental, en CVS no era así, y si en medio de un commit se cortaba la comunicación con el servidor, los ficheros quedaban a medio a subir, por así decir, de forma que unos aumentaban de versión y otros no, pero para el cliente loca, todos habián sido enviados y confirmados.
2. **Números de versión**: Destaca imperiosamente la atención, el hecho de que el versionado en CVS se mantega por fichero, mientras que en Subversion es global al todos, se habla, de revisión del proyecto.

Un repositorio Subversion está orientado a almacenar un proyecto, mientras un repositorio CVS está orientado a albergar varios proyectos, cada uno con sus números de versión.

3. **Almacenamiento de los ficheros del repositorio**: Mientras que en CVS los ficheros se guardan como archivos RCS (con extensión *,v*), que siempre podremos empaquetar en un tgz o un zip, en Subversion se almacenan en un base de datos, cuya robustez es al menos discutible.
4. **Conectividad al repositorio**: En este aspecto, Subversion se presenta claramente como un producto moderno y evolucionado frente a CVS. El hecho de que se pueda acceder al repositorio mediante http y https, le dá mayor flexibilidad que su predecesor.
5. **Estabilidad**: Subversion en este aspecto, se presenta como perdedor ante la robustez y madurez de CVS, sirva como ejemplo que hasta hace poco más de un año y medio, el propio Subversion se almacenaba en un repositorio CVS.

   Hoy por hoy, Subversion es un producto estable y perfectamente usable, aunque está por ver si soportará las cargas de trabajo y datos que CVS ha demostrado ser capaz de aguantar.
6. **Gestión de binarios**: El hecho de que Subversion detecte automáticamente los tipos de ficheros binarios supone una ventaja considerable frente a CVS. Internamente realiza diffs binarios para el versionado, optimizando el espacio según los propios desarrolladores me aseguraron. Las versiones de los ficheros binarios en CVS se guardan íntegramente, con lo que el repositorio crece desmesuradamente.
7. **Instalación y configuración**: Si quisiéramos usar Subversion como CVS, esto es accediendo al repositorio vía svn o svn+ssh, tendríamos en principio los mismos problemas con uno que con otro, pero obviamente, cuando queremos instalar Subversion, normalmente queremos que el repositorio esté accesible por http, con lo que se complica bastante la infraestructura frente a CVS, dado que debemos instalar Apache2 y configurarlo.
8. **Gestión de usuarios**: La gestión de usuarios en Subversion está mucho más evolucionada que en su homólogo CVS, implementando un sistema de listas de control de acceso de forma nativa, que nos permite dar visibilidad total o parcial, con permisos de lectura/escritura a cualquier subdirectorio o archivo de nuestro repositorio.
9. **Metainformación y autoTags**: Aunque en Subversion se le ha dado gran importancia a la metainformación asociada a cada archivo, y cada modificación sobre el mismo, decepciona el hecho de que haya que activarla manualmente, con los consiguientes problemas que ello acarrea en los ficheros binarios, y que la metainformación que nosotros creamos para un archivo concreto, no se expanda automáticamente.

   En principio, los usuarios de CVS no percibimos ninguna mejora sobre esto, si no más bien una molestia.
10. **Administración de ficheros**: Subversion soluciona con creces la gran carencia de comandos que tenía CVS a la hora de mover, renombrar y borrar archivos, conservando la historia de los mismos, que siempre había que hacerlo como administrador del sistema con el consiguiente riesgo que ello suponía.
11. **Versionado de la configuración**: La configuración del repositorio CVS estaba mantenida en el propio repositorio dentro del directorio CVSROOT. En Subversion se echa de menos esa centralización de la configuración, de forma que ello se delega a los propios clientes, perdiendo el versionado de la misma.

Estas diferencias deberían ser tenidas en cuenta a la hora de tomar una decisión sobre cuál de ellos usar, dado que una vez que se empieza a desarrollar sobre un sistema de control de versiones, puede resultar fatídico el migrar a otro sistema.

En general, el criterio que yo sigo cuando debo asesorar a alguien es:

1. **Implanta el que conozcas**.
2. Si las **entregas al repositorio se harán desde una intranet** donde existe alguien técnicamente preparado para solventar los problemas que aparezcan: **usa CVS**.

   Si por el contrario, se necesita **acceder al repositorio desde internet y una intranet, usa Subversion**.