UNIVERSIDAD DE CÁDIZ

ADMINISTRACIÓN DE SISTEMAS OPERATIVOS

Sistemas para el Control de Versiones

Rosa Mª Durante Lerate
Pablo Recio Quijano

Leandro Pastrana González
Noelia Sales Montes

Índice general

Capítulo 1

Introducción a los Sistemas de Control de Versiones

1.1. ¿Qué es un Sistema de Control de Versiones?

Un Sistema de Control de Versiones (en adelante **SCV**), es un software que controla y organiza las distintas **revisiones** que se realizen sobre uno o varios documentos.
Pero, ¿qué es una revisión? Se podría decir que una revisión es un cambio realizado sobre un documento, por ejemplo añadir un parrafo, borrar un fragmento o algo similar. Veamos un ejemplo:

Supongamos que cargamos en un SCV el siguiente código fuente:

```
#include <iostream>

int main(){
  cout << "Hola, mundo!" << endl;
}
```

Se añade al SCV como la revisión 1 del fichero. Una vez añadido, vemos que no compila, ya que nos falta incluir el uso del espacio de nombres, así que lo modificamos:

```
#include <iostream>

using namespace std;

int main(){
  cout << "Hola, mundo!" << endl;
}
```

Se vuelve a añadir al SCV, ahora como la revisión número 2. De esta forma, se guarda el **historial** de las distintas modificaciones sobre un fichero, por lo que en cualquier momento podemos restaurar la revisión que queramos de un fichero.

Esto presenta varias ventajas, aunque la principal y la más llamativa es que nos permite mantener una copia de seguridad de todas las modificaciones realizadas sobre un fichero, lo cual nos facilita la tarea de deshacer algo que esté mal. Supongamos por ejemplo que modificamos un proyecto software, y modificamos un módulo para arreglar un bug.
Ahora funciona para ese bug, pero nos hemos "cargado" otra funcionalidad más importante, que no

queremos perder. Simplemente volvemos a una revisión anterior, y no hay problemas.

1.2. ¿Para qué sirve un Sistema de Control de Versiones?

De forma general, se utilizan SCV para desarrollo de proyectos Software, sobre todo para realizar en grupos de desarrollo. Vamos a analizar un poco los motivos de porque esto es así.

1.2.1. Ejemplo de grupo de desarrollo sin Sistema de Control de Versiones

Supongamos un grupo de 5 desarrolladores, que están realizando un proyecto. Puede ser que el proyecto esté muy claro qué partes tiene que hacer cada uno, pero esto no suele ser así.

Estos desarrolladores van subiendo a un servidor tipo FTP las distintas modificaciones que realizan sobre el código. Uno sube un fichero fuente `claseA.h`, otro `entrada-salida.cpp`, etc...
Sin embargo, llegará un momento en el que dos desarrolladores modifiquen el mismo fichero. Veamos un ejemplo claro, con el siguiente código:

```
#include <iostream>

using namespace std;

int main(){
  int *v = {1,2,3,5,9,10,3,2}; // esto no es correcto para el compilador,
                               // pero nos vale conceptualmente.
  int a = 7 ,b = 3;

  // -------- Swap entre a y b
  int aux = a;
  a = b;
  b = aux;
  //--------------------------
  //------- Maximo de *v -----
  int max = v[0];
  for (int i = 0; i < 8 ; i++){
    if (v[i] > max){
      max = v[i];
    }
  }
  //--------------------------

  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
  cout << "max de v = " << max << endl;
}
```

Ahora, el desarrollador Pepe decide modular algo más el código y añadir una función que realize el swap de dos variables enteras, quedando el siguiente código:

```
#include <iostream>

using namespace std;

void swap(int &a, int &b);
```

```
6
7  int main(){
8    int *v = {1,2,3,5,9,10,3,2}; // esto no es correcto para el compilador,
9                                 // pero nos vale conceptualmente.
10   int a = 7 ,b = 3;
11
12   // -------- Swap entre a y b
13   swap(a,b);
14   //--------------------------
15   //------- Maximo de *v -----
16   int max = v[0];
17   for (int i = 0; i < 8 ; i++){
18     if (v[i] > max){
19       max = v[i];
20     }
21   }
22   //--------------------------
23
24   cout << "a = " << a << endl;
25   cout << "b = " << b << endl;
26   cout << "max de v = " << max << endl;
27 }
28
29 void swap(int &a, int &b){
30   int aux = a;
31   a = b;
32   b = aux;
33 }
```

Realiza el cambio y sube el fichero al servidor FTP, “machacando” el que ya estaba ahí. Ahora resulta que en el intervalo de tiempo en el que el desarrollador Pepe estaba realizando la función `swap()`, la desarrolladora `María` decide realizar una función que localize el máximo de un vector y lo devuelva, ya que considera que se puede utilizar en muchos más sitios:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int maximo(int *v, int tam);
6
7  int main(){
8    int *v = {1,2,3,5,9,10,3,2}; // esto no es correcto para el compilador,
9                                 // pero nos vale conceptualmente.
10   int a = 7 ,b = 3;
11
12   // -------- Swap entre a y b
13   int aux = a;
14   a = b;
15   b = aux;
16   //--------------------------
17   //------- Maximo de *v -----
18   int max = maximo(v,8);
19   //--------------------------
20
21   cout << "a = " << a << endl;
```

```
22    cout << "b = " << b << endl;
23    cout << "max de v = " << max << endl;
24  }
25
26  int maximo(int *v, int tam){
27    int max = v[0];
28    for (int i = 0 ; i < tam ; i++){
29      if (v[i] > max){
30        max = v[i];
31      }
32    }
33  }
```

Ella descargó la revisión anterior a la modificación de Pepe, por lo que sube el fichero y "machaca" la versión anterior, que es la que había hecho Pepe, eliminando su trabajo, el cual además era complementario al de María. Por tanto la próxima vez que Pepe compruebe el servidor, verá que lo que hizo ya no está.

Aquí entran en juego los sistemas de control de versiones. Si el grupo de desarrollo usara un SCV, esto no pasaría, ya que estos sistemas vigilan sobre que líneas se han hecho estas modificaciones, de forma que "verían" que los dos cambios no son incompatibles, y la nueva versión *mergerá* (del inglés "merging") las dos modificaciones.
Incluso sin esta función, el problema de conflicto, sería más facilmente resoluble, ya que al mantener versiones anteriores se pueden rescatar, para ver que cambios se han hecho y realizar esa unión de forma manual, aunque si es automatizable, mejor.

Extrapolemos esto a cualquier desarrollo de software con un grupo relativamente grande, como por ejemplo los desarrolladores de cualquier proyecto de Software Libre medianamente grande, en el cual puede haber decenas de personas distintas implicadas en el desarrollo. Si fuera un grupo como el descrito arriba, el desarrollo puede llegar a ser un auténtico caos. Por este motivo se hace indispensable el uso de alguna herramientas de SCV en desarrollos de software mayores que los unipersonales.
Aunque no está restringido solo al software, ya que trabaja con cualquier tipo de ficheros de texto plano, por lo que también se puede usar para generar documentación de forma colaborativa usando LaTeX por ejemplo[1].

1.2.2. Ejemplos en el Software Libre

Hemos visto antes que fundamentalmente el uso de SCV está destinado al desarrollo de Software, ayudando a la sincronización de los distintos miembros del grupo de desarrollo. Esto se cumple tanto en entornos privados (empresas) como en entornos públicos (proyectos libres). Veamos algunos ejemplos de proyectos libres que utilizan estas herramientas:

Kernel de Linux: Utiliza GIT desde el año 2005. Previamente utilizaba BITKEEPER.

KDE: Utiliza SUBVERSION. A fecha del 2 de Mayo de 2009, andan por la revisión 962574, cerca de un millón, habiendo sido la primera en el año 1997. Actualmente trabajan en el proyecto más de 200 personas. ¿Imaginamos como sería sincronizar todas estas personas sin un SCV? Es totalmente imposible.

Firefox: Utilizan CVS.

Ubuntu: Utiliza BAZAAR, un sistema distribuido.

[1] Los profesores del Departamento de Estadística, lo usan para el libro de R.

Podríamos continuar la lista con muchísimos más. Por ejemplo, los sistemas de forjas[2] dan soporte para SCV, incluyendo herramientas para navegar en el árbol del repositorio.

1.3. Tipos de Sistemas de Control de Versiones

Fundamentalmente, podemos distinguir dos tipos distintos de SCV: **centralizados** y **distribuidos**.

1.3.1. Sistemas centralizados

Presentan la característica fundamental de que funcionan como un entorno clásico *Cliente- Servidor*. Es decir, tendremos un servidor en el que se alojará el repositorio del proyecto, con toda la información de los cambios, ficheros binarios añadidos, ...
En estos sistemas, el cliente trabaja con una "copia de trabajo" del servidor, la cual es realmente una copia de como estaba el servidor en una revisión determinada - normalmente es la más actualizada. El desarrollador hace cambios sobre esa copia de trabajo, y cuando considera que ha terminado con esa modificación la sube (`commit`) al servidor, el cual se encargará de fundir esos cambios en el repositorio central, resolver conflictos si pudiera, ó informar al usuario de los errores que se hayan podido dar.
Además, estos sistemas pueden a su vez dividirse en dos tipos más, según la forma que tengan de controlar los posibles conflictos sobre un mismo fichero desde más de un cliente:

Bloqueo del archivo

Aplicando el principio de exclusión mutua, lo que hacen estos sistemas es muy simple: cuando alguien está modificando un archivo, bloquea el acceso de escritura a ese archivo para el resto de usuarios.
Esto implica que habrá menos conflictos cuando se fundan distintas ramas, pero presentan una enorme cantidad de problemas extra:

- Alguien puede "olvidarse" de abrir el archivo para el resto.
- Puede ocasionar que alguien trabaje de forma local para evitar el cerrojo, y a la hora de fundir los cambios sea aun peor.

Fusión de versiones

Es el usado en la mayoría de los SCV. Es lo que se explicó en el ejemplo anterior: el SCV controla qué líneas del código se han cambiado en cada revisión, por lo que si dos desarrolladores cambian zonas distintas de un mismo fichero, el sistema podrá fundir ambos cambios en la versión del servidor.

Sin embargo, estos sistemas pueden fallar por un simple indentado, una línea de más o de menos, y en muchas ocasiones es el programador quien debe corregir esos conflictos a mano, que en muchos casos es bastante problemático.

1.3.2. Sistemas distribuidos

Si los sistemas centralizados utilizan un modelo clásico de entorno *cliente-servidor*, se podría decir que un sistema distribuido es similar a un sistema *Peer-to-Peer (P2P)*.
En estos sistemas, en lugar de que cada cliente tiene una copia de trabajo del (único) servidor, la copia de trabajo de cada cliente es un repositorio en sí mismo, una rama nueva del proyecto central. De esta forma, la sincronización de las distintas ramas se realiza intercambiando "parches" con otros clientes

[2] Como por ejemplo: `sourceforge.net`, `forja.rediris.es`, ...

del proyecto. Esto es claramente un enfoque **muy** diferente al de los sistemas centralizados, por diversos motivos:

- No hay una copia original del código del proyecto, solo existen las distintas copias de trabajo.
- Operaciones como los commits, mirar el historial o rehacer cambios, no necesitan de una conexión con un servidor central, esta conexión solo es necesaria al "compartir" tu rama con otro cliente del sistema.
- Cada copia de trabajo es una copia remota del código fuente y de la historia de cambios, dando una seguridad muy natural contra la pérdida de los datos.

Capítulo 2

Funcionamiento para el cliente

2.1. Introducción al cliente en un SCV

Todos los sistemas de control de versiones se basan en disponer de un **repositorio**, que es el conjunto de información gestionada por el sistema. Este repositorio contiene el historial de versiones de todos los elementos gestionados. Con lo cual, lo primero que debe hacer el cliente es obtener su copia de trabajo del repositorio; dependiendo de que tipo de **Sistema de Control de Versiones** utilice se obtendrá la copia de trabajo de una forma u otra.

A partir de este momento ya el cliente podrá realizar sus aportes al repositorio. Para ello, el cliente trabajara bajo su copia de trabajo y realizará las modificaciones oportunas. Sin embargo, esas modificaciones no se tendrán en cuenta en el repositorio hasta que no se envíen al él.

Aunque los tipos de **SCV** son muy distintos entre ellos, desde el punto de vista del cliente no lo son tanto: el hecho de que el repositorio se organice de forma distribuida o centralizada no afecta al estilo de trabajo del cliente.

2.1.1. Términos básicos

Repositorio Es el lugar donde se almacenan los datos actualizados e históricos en un servidor.

Módulo Conjunto de directorios y/o archivos dentro del repositorio que pertenecen a un proyecto común.

Rotular (`tag`) Darle a alguna versión de cada uno de los ficheros del módulo en desarrollo en un momento preciso un nombre común ("etiqueta" o "rótulo") para asegurarse de reencontrar ese estado de desarrollo posteriormente bajo ese nombre.

Versión Una revisión es una versión determinada de un archivo.

Ramificar (`branch`) Un módulo puede ser bifurcado en un momento de tiempo de forma que, desde ese momento en adelante, dos copias de esos ficheros puedan ser desarrolladas a diferentes velocidades o de diferentes formas, de modo independiente.

Desplegar (`check-out`) Un despliegue crea una copia de trabajo local desde el repositorio. Se puede especificar una revisión concreta, y por defecto se suele obtener la última.

Envío (`commit`) Una copia de los cambios hechos a una copia local es escrita o integrada sobre repositorio.

Conflicto Puede ocurrir en varias circunstancias:

- Los usuarios X e Y despliegan versiones del archivo A en que las líneas n1 hasta n2 son comunes.
- El usuario X envía cambios entre las líneas n1 y n2 al archivo A.
- El usuario Y no actualiza el archivo A tras el envío del usuario X.
- El usuario Y realiza cambios entre las líneas n1 y n2.
- El usuario Y intenta posteriormente enviar esos cambios al archivo A.

En estos casos el sistema es incapaz de fusionar los cambios y pide al usuario que sea él quien **resuelva** el conflicto.

Resolver El acto de la intervención del usuario para atender un conflicto entre diferentes cambios al mismo documento.

Cambio (`diff`) Un cambio representa una modificación específica a un documento bajo control de versiones.

Lista de Cambios (`changelist`) En muchos sistemas de control de versiones con `commits` multi-cambio atómicos, una lista de cambios identifica el conjunto de cambios hechos en un único `commit`.

Exportación Una exportación es similar a un check-out, salvo porque crea un árbol de directorios limpio sin los metadatos de control de versiones presentes en la copia de trabajo.

Importación Una importación es la acción de copia un árbol de directorios local (que no es en ese momento una copia de trabajo) en el repositorio por primera vez.

Fusión (`merge`) Una integración o fusión une dos conjuntos de cambios sobre un fichero o un conjunto de ficheros en una revisión unificada de dicho fichero o ficheros. Puede suceder cuando:

- Un usuario, trabajando en esos ficheros, actualiza su copia local con los cambios realizados, y añadidos al repositorio, por otros usuarios. Análogamente, este mismo proceso puede ocurrir en el repositorio cuando un usuario intenta subir sus cambios.
- Después de que el código haya sido ramificado, un problema anterior a dicha separación sea arreglado en una rama y se necesite incorporar dicho arreglo en la otra.
- Después de que los ficheros hayan sido ramificados, desarrollados de forma independiente por un tiempo, y que entonces se haya requerido que fueran fundidos de nuevo en una única rama unificada.

Integración inversa El proceso de fundir ramas de diferentes equipos en el trunk principal del sistema de versiones.

Actualización (`update`) Una actualización integra los cambios que han sido hechos en el repositorio (por ejemplo por otras personas) en la copia de trabajo local.

Copia de trabajo La copia de trabajo es la copia local de los ficheros de un repositorio, en un momento del tiempo o revisión específicos. Todo el trabajo realizado sobre los ficheros en un repositorio se realiza inicialmente sobre una copia de trabajo, de ahí su nombre. Conceptualmente, es un cajón de arena o sandbox.

Congelar Significa permitir los últimos cambios (`commits`) para solucionar las fallas a resolver en una entrega (`release`) y suspender cualquier otro cambio antes de una entrega, con el fin de obtener una versión consistente.

2.2. Ciclo de trabajo con SCV centralizado: SUBVERSION

Cuando un cliente quiere comenzar a desarrollar un proyecto utilizando SUBVERSION primeramente debe instalar la herramienta de la siguiente forma:

```
sudo apt-get install subversion
```

Cuando ya tiene la herramienta lista para utilizar ejecutará el `checkout` inicial:

```
svn checkout http://ruta/repositorio
```

Por ejemplo:

```
svn checkout http://192.168.2.101/svn/RepositorioEjemplo
```

A partir de este momento el cliente ya tiene en su poder una copia de trabajo del repositorio en el cual podrá realizar sus modificaciones e incluirlas en el sistema de control de versiones.
El ciclo de trabajo que se debe llevar es el siguiente:

- Actualizar la copia de trabajo (por si otros compañeros han realizado modificaciones):

  ```
  svn update
  ```

- Examinar los cambios a partir de comentarios en las revisiones. Cada vez que se envían las modificaciones al servidor se incluye un comentario sobre qué es lo que se ha modificado para esa revisión. Estos comentarios son los *logs*. Para ver los logs se introduce en consola la siguiente línea de comandos:

  ```
  svn log -r revmenor:revmayor
  ```

  De esta forma se mostrará en pantalla los comentarios de cada revisión para informarnos de las modificaciones que se han realizado en ellas. Esto es muy útil ejecutarlo antes de ponernos a modificar para informarnos de cómo va el proyecto.

- Realizar los cambios pertinentes a los ficheros y directorios: mover, copiar, borrar, crear... (por ahora serán cambios locales). Hay que hacerlos con la asistencia de SUBVERSION. De esta forma se mantendrá el historial de cambios.
  Se utiliza de la siguiente forma:

  ```
  svn add [fichero|directorio]
  svn mkdir directorio
  svn delete [fichero|directorio]
  svn mv [fich|dir]Original [fich|dir]Cambiado
  svn cp [fich|dir]Original [fich|dir]Copiado
  ```

  Veamos varios ejemplos de uso de estos comandos:

  ```
  # Creamos fichero1.c
  svn add fichero1.c
  # Creamos fichero2.c
  ```

```
svn add fichero2.c
# Creamos dirficheritos
svn mkdir dirficheritos
# Copiamos el fichero
svn cp fichero1.c ficheritos/fichero1.h
# Movemos el fichero
svn mv fichero2.c ficheritos/fichero1.cpp
# Borramos fichero1.c
svn delete fichero1.c
```

- Examinar los cambios realizados. Podemos comprobar el estado de los ficheros de la copia de trabajo de la siguiente forma:

```
svn status
```

  Este comando muestra todos los cambios realizados en la copia de trabajo desde que se realizó el último `svn update`. Puede funcionar sin red ya que sólo es necesario tener las carpetas ocultas *.svn*. Los códigos más comunes que nos muestra son:

  A fichero o directorio añadido.

  D fichero o directorio borrado.

  U fichero o directorio actualizado con cambios remotos.

  M fichero o directorio modificado localmente.

  G fichero o directorio acutalizado con cambios locales y remotos reunidos automáticamente.

  C los cambios locales y remotos han de ser reunidos localmente (conflicto en modificaciones).

  ? fichero o directorio que no se haya bajo el sistema de control de versiones y no se está ignorando.

  ! fichero o directorio falta en el sistema de control de versiones (puede haberse borrado localmente).

  $\sim$ el fichero o directorio no es del tipo esperado (se obtiene un directorio y se esperaba un fichero o viseversa).

  + se copiará información de historial además de contenido.

- Examinar las diferencias de ficheros entre dos revisiones. Ésto es útil cuando queremos enviar modificaciones cuando no tenemos acceso de escritura o para ver que han modificado los compañeros. Por defecto compara la revisión modificada localmente con la última registrada en el servidor. El comando a utilizar es:

```
svn diff [-r revmenor:revmayor] fichero
```

- Deshacer posibles cambios en un fichero o directorio:

```
svn revert fichero
```

  Esto restaura al fichero obtenido al del último `svn update`.

- Resolver conflictos. Es muy conveniente el realizar un `svn update` antes de subir nuestras modificaciones por si existiesen conflictos.

- Por último, para poder enviar las modificaciones al servidor usaremos el comando:

```
svn commit -m log_entre_comillas
```

  De esta forma quedarán almacenadas en el servidor las modificaciones realizadas en nuestra copia de trabajo además de la pequeña descripción que aportamos sobre éstas.

2.3. Ciclo de trabajo con SCV distribuido: GIT

Como en todos los sistemas de control de versiones tenemos que instalar la aplicación. Para GIT esto se hace con el siguiente comando:

```
sudo apt-get install git-core
```

Para comenzar con GIT primero tenemos que introducir nuestros datos (nombre y correo electrónico) para que el sistema nos indique como autor en cada una de nuestras revisiones. Además podemos informar a GIT de cuál es nuestro editor favorito (por defecto usa Vim). Realizamos:

```
git config --global user.name "Nombre Apellidos"
git config --global user.email micorreo@ejemplo.com
git config --global user.editor mieditor
```

Para comenzar con el ciclo de trabajo tenemos que crear nuestro repositorio vacío, o clonar uno existente. También tenemos que tener en cuenta si podrá acceder al repositorio varios usuarios. Crearemos un directorio vacío para, a partir de él crear un repositorio vacío y que se pueda compartir con todos los miembros del grupo al que pertenezcan los ficheros del repositorio.

```
mkdir directorioVacio
git init --shared=group
```

Podemos observar que se ha creado un directorio oculto *.git*, que contiene los ficheros útiles para poder realizar todas las operaciones.
En la Sección 4.3.7 aclararemos qué ficheros son y qué contiene cada uno de ellos.

Para saber el estado de los ficheros en GIT tenemos el comando:

```
git status
```

Si en el directorio anterior creamos un fichero f cualquiera y ejecutamos el comando anterior obtenemos la siguiente salida:

```
# On branch master
#
# Initial commit
#
# Untracked files:
# (use "git add <file>..." to include in what
# will be committed)
#
# f
nothing added to commit but untracked files present
```

```
(use "git add" to track)
```

Con esto sabemos que:

- Nos encontramos en la rama principal del repositorio creado.
- La siguiente revisión que creamos será la primera.
- Existen algunos fichero cuyos cambios no están monitorizados (añadidos al repositorio). En nuestro caso, nos está avisando que faltaría añadir el fichero f.

Veamos pues cómo podemos añadir ficheros para prepararlos para envío.

```
git add f
```

También podemos borrar ficheros con la siguiente orden:

```
git rm fichero
```

Ahora enviaremos las modificaciones y crearemos la primera revisión con:

```
git commit -m comentario_de_la_revision
```

Hay que tener en cuenta que con GIT trabajamos con tres estructuras de datos: el repositorio (el directorio *.git*), el índice o caché (*.git/index*) y la copia de trabajo (todo lo que está fuera de *.git*).
Con lo cual podemos comparar qué estados tiene cada directorio y qué diferencias existen con la siguiente orden:

```
git diff
```

Obteniendo las diferencias entre el directorio de trabajo y la versión preparada para envío en el índice, además de destacar aquella línea que no preparemos para envío.

```
git diff --cached
```

Esta orde compara la versión del índice con la versión de la revisión actual (HEAD), y así nos señalará que versión tenemos preparada para enviar.
Para poder deshacer cambios en GIT podemos restaurar un fichero del directorio de trabajo a la versión que se halle en el índice de la siguiente forma:

```
git checkout fichero
```

Las formas más comunes de deshacer cambios en un fichero son:

ACCIÓN A DESHACER	AFECTA A	ORDEN
Añadido Modificado Borrado	Índice	`git reset -- f`
Añadido		`rm f`

Continúa en la página siguiente...

Continuación de la página anterior...		
ACCIÓN A DESHACER	AFECTA A	ORDEN
Modificado	Directorio de Trabajo	`git checkout f`
Borrado		`git checkout f`
Añadido		`git rm -f f`
Modificado	Ambos	`git checkout HEAD f`
Borrado		`git checkout HEAD f`

Tabla 2.1: Órdenes para deshacer acciones

Si no hemos enviado aún nuestros cambios a ningún otro repositorio ni nadie nos ha tomado de nuestro repositorio podemos corregir errores fácilmente en la última revisión que hayamos hecho:

```
git commit --amend
```

Con esto la anterior revisión se ignorará y será efectivamente sustituida por otra que incorporará sus cambios y los que hayamos hecho ahora.
Para poder etiquetar una revisión ejecutamos:

```
git tag etiqueta
```

También podemos etiquetar cualquier otra revisión creando objetos reales de tipo "etiqueta" con comentarios (-a) y firmada con nuestra clave pública GnuPG (-s) que tenga el mismo correo y nombre que le dimos a GIT:

```
  git tag -a -s etiqueta revision
```

Finalmente podemos redistribuir los contenidos de cualquier revisión mediante la orden :

```
git archive etiqueta_de_la_revision
```

Capítulo 3

El servidor en Sistemas Centralizados

3.1. Introducción a los Sistemas de Control de Versiones Centralizados

En un sistema de control de versiones centralizado todos los ficheros y sus versiones están almacenados en un único directorio de un un servidor.

Todos los desarrolladores que quieran trabajar con esos fuentes deben pedirle al sistema de control de versiones una copia local para poder añadir o modificar ficheros. En ella realizan todos sus cambios y una vez hechos, se informa al sistema de control de versiones para que guarde los fuentes modificados como una nueva versión.

Es decir, un sistema de control de versiones centralizado funciona según el paradigma clásico *cliente-servidor*.

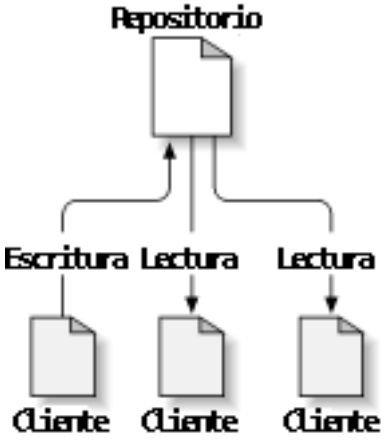

Figura 3.1: Sistema Cliente-Servidor Típico

Una vez subido el código, está disponible para otros desarrolladores, que pueden actualizar la información del repositorio y solicitar la nueva versión. Para el intercambio de fuentes entre ellos, es necesario pasar por el repositorio de fuentes del servidor. Por tanto, en el servidor están guardadas todas las versiones y los desarrolladores sólo tienen en su copia local aquellos ficheros que han solicitado del servidor.

Los sistemas de control de versiones centralizados libres más conocidos son CVS y SUBVERSION, de los que hablaremos a continuación, aunque nos centraremos más en este último, puesto que es el más actual de los dos.

3.2. El enfoque Centralizado

Cuando se desarrollaron los sistemas de control de versiones, muchos de ellos seguían el esquema cliente-servidor, con lo que el desarrollo de las herramientas está claramente diferenciado entre dos elementos clave:

- el cliente, que es la aplicación que se conecta al servidor y mantiene una copia local del repositorio, así como se encarga, generalmente de la corrección de conflictos y la mayoría de lógica del sistema.
- el servidor, que es el sistema que toma los datos del cliente y los almacena a espera de una o múltiples peticiones de esos datos. Este sistema se encargaría del almacenamiento de las versiones, control de concurrencia, bloqueos y otras características parecidas a las de las bases de datos.

Este desarrollo conlleva algunas características clave (tanto positivas como negativas), entre las cuales podemos adelantar las siguientes:

- El sistema servidor es un repositorio, como los que mantienen los clientes, pero perfectamente sincronizado y sin que dé lugar a conflictos.
  Dicho de otro modo: es la copia maestra de los datos.
- Cuando un sistema web quiere hacer un listado, puede tomar los datos de este servidor y siempre serán fiables, con lo que no tendrá que resolver conflictos ni incongruencias.
- Una copia local debe poder mezclarse con el repositorio central cuando queramos publicar un conjunto de cambios o cuando queramos tomar la última versión publicada en concordancia con nuestra copia local.
- Es lógico que en desarrollo de software aparezcan ramificaciones, versiones, etiquetas, o similares, a modo de tener varias copias de (secciones del) proyecto según nos interese.
  Estas ramificaciones están en el servidor y en algunos casos puede llegar a ser muy costosa su diferenciación.

3.3. CVS

3.3.1. Introducción

El *Concurrent Versions System* (también conocido como *Concurrent Versioning System*) es una aplicación informática bajo licencia GPL que implementa un sistema de control de versiones.

Se encarga de mantener el registro de todo el trabajo y los cambios en los ficheros (código fuente principalmente) que conforman un proyecto y permite que distintos desarrolladores (potencialmente situados a gran distancia) colaboren.

3.3.2. Características

Como sistema de control de versiones centralizado, CVS utiliza una arquitectura cliente-servidor como el descrito en la introducción.

Típicamente, cliente y servidor se conectan utilizando Internet, pero con el sistema CVS el cliente y servidor pueden estar en la misma máquina. El sistema CVS tiene la tarea de mantener el registro de la historia de las versiones del programa de un proyecto solamente con desarrolladores locales.

Originalmente, el servidor utilizaba un sistema operativo similar a Unix, aunque en la actualidad existen versiones de CVS en otros sistemas operativos, por lo que los clientes CVS pueden funcionar en cualquiera de los sistemas operativos más difundidos.

Varios clientes pueden sacar copias del proyecto al mismo tiempo. Posteriormente, cuando actualizan sus modificaciones, el servidor trata de acoplar las diferentes versiones. Si esto falla, por ejemplo debido a que dos clientes tratan de cambiar la misma línea en un archivo en particular, entonces el servidor deniega la segunda actualización e informa al cliente sobre el conflicto, que el usuario deberá resolver manualmente.
Si la operación de ingreso tiene éxito, entonces los números de versión de todos los archivos implicados se incrementan automáticamente, y el servidor CVS almacena información sobre la actualización, que incluye una descripción suministrada por el usuario, la fecha y el nombre del autor y sus archivos log.

Entre otras funcionalidades, los clientes pueden también comparar diferentes versiones de archivos, solicitar una historia completa de los cambios, o sacar una "foto"histórica del proyecto tal como se encontraba en una fecha determinada o en un número de revisión determinado.

Los clientes también pueden utilizar la orden de actualización con el fin de tener sus copias al día con la última versión que se encuentra en el servidor. Esto elimina la necesidad de repetir las descargas del proyecto completo.

CVS también puede mantener distintas ramas"de un proyecto. Por ejemplo, una versión difundida de un proyecto de programa puede formar una rama y ser utilizada para corregir errores. Todo esto se puede llevar a cabo mientras la versión que se encuentra actualmente en desarrollo y posee cambios mayores con nuevas características se encuentre en otra línea formando otra rama separada.

3.3.3. Estado actual del proyecto

CVS fue desarrollado por GNU, el sitio GNU distribuye el programa, denominándolo "paquete GNUçon aplicaciones básicas a través de esta página. En otros proyectos se otorga con licencia GPL.

Actualmente existen muchos derivados de CVS implantados en los diferentes sistemas operativos.

3.3.4. Limitaciones del protocolo CVS

- Los archivos en el repositorio sobre la plataforma CVS no pueden ser renombrados: éstos deben ser agregados con otro nombre y luego eliminados.
- Soporte limitado para archivos Unicode con nombres de archivo no ASCII.

3.3.5. Requisitos mínimos

CVS requiere de la instalación de dos aplicaciones:

- Por un lado un servidor instalado en un ordenador.
  Este servidor es el que almacena todas las versiones de todos los ficheros.
- Por otro lado, en cada computador de trabajo necesitamos un cliente de CVS.
  Este es un programa capaz de conectarse con el servidor y pedirle o enviarle ficheros fuentes, según se solicite.

No entraremos en detalles de instalación ni configuración de CVS, puesto que el sistema está actualmente en desuso. Nos centraremos más en el mismo apartado para el sistema de control de versiones SUBVERSION.

3.4. SUBVERSION

SVN es conocido así por ser el nombre del cliente, el software en sí es llamado SUBVERSION.

SUBVERSION es un sistema centralizado para compartir información que fue diseñado como reemplazo de CVS.

La parte principal de SUBVERSION es el repositorio, el cual es un almacén central de datos. El repositorio guarda información en forma de árbol de archivos. Un número indeterminado de clientes puede conectarse al repositorio para leer (el cliente recibe información de otros) o escribir (un cliente pone a disposición de otros la información) en esos archivos.

3.4.1. Ventajas

- Se sigue la historia de los archivos y directorios a través de copias y renombrados.
- Las modificaciones (incluyendo cambios a varios archivos) son atómicas.
- La creación de ramas y etiquetas es una operación más eficiente.
  Tiene coste de complejidad constante ($O(1)$) y no lineal ($O(n)$) como en CVS.
- Se envían sólo las diferencias en ambas direcciones (en CVS siempre se envían al servidor archivos completos).
- Puede ser servido mediante Apache, sobre WebDAV/DeltaV. Esto permite que clientes WebDAV utilicen SUBVERSION en forma transparente.
- Maneja eficientemente archivos binarios (a diferencia de CVS que los trata internamente como si fueran de texto).
- Permite selectivamente el bloqueo de archivos. Se usa en archivos binarios que, al no poder fusionarse fácilmente, conviene que no sean editados por más de una persona a la vez.
- Cuando se usa integrado a Apache permite utilizar todas las opciones que este servidor provee a la hora de autentificar archivos (SQL, LDAP, PAM, etc.).

3.4.2. Carencias

- El manejo de cambio de nombres de archivos no es completo. Lo maneja como la suma de una operación de copia y una de borrado.
- No resuelve el problema de aplicar repetidamente parches entre ramas, no facilita el llevar la cuenta de qué cambios se han trasladado. Esto se resuelve siendo cuidadoso con los mensajes de commit. Esta carencia será corregida en la próxima versión (1.5).

3.4.3. Arquitectura de SUBVERSION

La siguiente imagen ilustra la arquitectura básica de un sistema general basado en SUBVERSION:

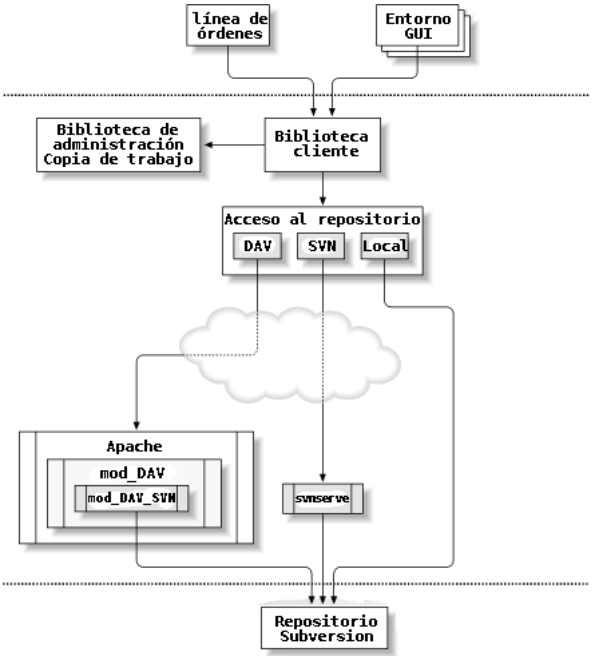

Figura 3.2: Arquitectura de SUBVERSION

En un extremo se encuentra un repositorio de SUBVERSION que conserva todos los datos versionados. Al otro lado, hay una aplicación cliente SUBVERSION que administra las réplicas parciales de esos datos versionados (llamadas *"copias de trabajo"*).

Entre estos extremos hay múltiples rutas a través de varias capas de acceso al repositorio (AR). Algunas de estas rutas incluyen redes de ordenadores y servidores de red que después acceden al repositorio. Otras pasan por alto la red y acceden al repositorio directamente.

3.4.4. Componentes de SUBVERSION

Una vez instalado, SUBVERSION se compone de varios elementos o aplicaciones, entre las cuales tenemos:

svn El programa cliente de línea de comandos.

svnversion Programa para informar del estado de una copia de trabajo.

svnlook Una herramienta para inspeccionar un repositorio de SUBVERSION.

svnadmin Herramienta para crear, modificar o reparar un repositorio de SUBVERSION.

svndumpfilter Un programa para filtrar el formato de salida de volcado de repositorios SUBVERSION.

mod_dav_svn Un módulo para el servidor HTTP Apache usado para hacer que su repositorio esté disponible a otros a través de una red.

svnserve Un servidor independiente, ejecutable como proceso demonio o invocable por SSH; otra manera de hacer que su repositorio esté disponible para otros a través de una red.

3.4.5. Copias de trabajo

Hemos hablado en varias ocasiones de las copias de trabajo, sin embargo, no hemos detallado qué hay detrás de estas copias de trabajo en el entorno del cliente.

Una copia de trabajo contiene, además del árbol de directorios y ficheros almacenados en el repositorio, algunos archivos extra, creados y mantenidos por SUBVERSION para simplificar la ejecución de las órdenes a nivel de cliente.

En particular, cada directorio de una copia de trabajo contiene un subdirectorio llamado *.svn*, también conocido como el directorio administrativo de la copia de trabajo. Los archivos en cada directorio administrativo ayudan a SUBVERSION a reconocer qué archivos contienen cambios no publicados y qué archivos están desactualizados con respecto al repositorio del servidor.

Un repositorio típico de SUBVERSION contiene a menudo los archivos (o el código fuente) de varios proyectos; normalmente, cada proyecto es un subdirectorio en el árbol del sistema de archivos del repositorio. En esta disposición, la copia de trabajo de un usuario se corresponde habitualmente con un subárbol particular del repositorio.

Por ejemplo, podemos suponer un repositorio que contiene dos proyectos de software, *proyecto1* y *proyecto2*, donde cada uno reside en su propio subdirectorio dentro del directorio raíz.

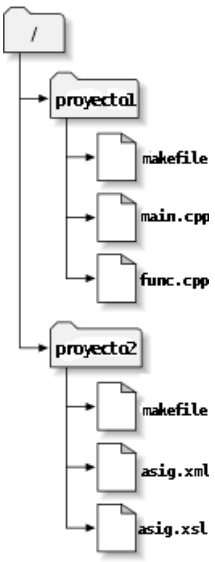

Figura 3.3: Esquema de un repositorio típico

El directorio SVN

Dentro de cada subdirectorio de una copia de trabajo se encuentra un directorio SVN, el cual contiene toda la información que se necesita para realizar las operaciones del protocolo SVN y además poder ejecutarlas en un tiempo eficiente.

Este directorio se denota como *.svn* y su contenido es el siguiente:

- `.svn/` ⟶ Directorio SVN
- `.svn/format` ⟶ Formato del directorio
- `.svn/all-wcprops` ⟶ Propiedades de todos los elementos que contiene el directorio

⋆ `.svn/prop-base/` ⟶ Directorio con las propiedades iniciales (backup) de cada fichero modificado

⋆ `.svn/prop-base/fichero-modificado.svn-base` ⟶ Fichero que contiene las propiedades del fichero modificado antes de ser variado

⋆ `.svn/text-base/` ⟶ Directorio con todos los ficheros modificado tal y como se encontraban antes de ser modificados

⋆ `.svn/text-base/fichero-modificado.svn-base` ⟶ Backup del fichero modificado en su estado anterior (tal y como se encuentra en el repositorio del servidor)

⋆ `.svn/entries` ⟶ Modificaciones que se han de tener en cuenta al realizar el commit

⋆ `.svn/props/` ⟶ Directorio con las propiedades (generalmente vacío)

⋆ `.svn/tmp/` ⟶ Ficheros temporales

⋆ `.svn/tmp/prop-base/` ⟶ Ficheros temporales

⋆ `.svn/tmp/text-base/` ⟶ Ficheros temporales

⋆ `.svn/tmp/props/` ⟶ Ficheros temporales

3.4.6. URLs del repositorio

A los repositorios de SUBVERSION se puede acceder a través de diferentes métodos en el disco local, o a través de varios protocolos de red. Sin embargo, la ubicación de un repositorio es siempre un URL.

A continuación podemos observar la correspondencia entre los diferentes esquemas de URL y los métodos de acceso disponibles.

ESQUEMA	MÉTODO DE ACCESO
file : ///	acceso directo al repositorio (en disco local)
http : //	acceso vía protocolo WebDAV a un servidor Apache que entiende de SUBVERSION
https : //	igual que http://, pero con cifrado SSL
svn : //	acceso vía un protocolo personalizado a un servidor svnserve
svn+ssh : //	igual que svn://, pero a través de un túnel SSH

Tabla 3.1: URLs de Acceso al Repositorio

En general, los URLs de SUBVERSION utilizan la sintaxis estándar, permitiendo la especificación de nombres de servidores y números de puertos como parte del URL.

NOTA: Recuerde que el método de acceso *file:* es válido sólo para ubicaciones en el mismo servidor donde se ejecuta el cliente. De hecho, se requiere por convención que la parte del URL con el nombre del servidor esté ausente o sea *localhost*:

```
$ svn checkout file:///ruta/a/repositorio
...
$ svn checkout file://localhost/ruta/a/repositorio
```

Además, los usuarios del esquema *file:* en plataformas Windows necesitarán usar una sintaxis "estándar" extraoficial para acceder a repositorios que están en la misma máquina, pero en una unidad de disco distinta de la que el cliente esté utilizando en el momento. Cualquiera de las dos siguientes sintaxis para rutas de URL funcionarán siendo X la unidad donde reside el repositorio:

```
C:\> svn checkout file:///X:/ruta/a/repositorio
...
C:\> svn checkout "file:///X|/ruta/a/repositorio"
```

3.4.7. Revisiones

En una copia privada del repositorio, un cliente puede cambiar el contenido de los ficheros, crear, borrar, renombrar y copiar ficheros y directorios, y luego enviar el conjunto entero de cambios como si se tratara de una unidad.

En el repositorio, cada cambio es tratado como una transacción atómica: o bien se realizan todos los cambios, o no se realiza ninguno. SUBVERSION trata de conservar esta atomicidad para hacer frente a posibles fallos del programa, fallos del sistema, problemas con la red, y otras acciones del usuario.

Cada vez que el repositorio acepta un envío, éste da lugar a un nuevo estado del árbol de ficheros llamado revisión. A cada revisión se le asigna un número natural único, una unidad mayor que el número de la revisión anterior. La revisión inicial de un repositorio recién creado se numera con el cero, y consiste únicamente en un directorio raíz vacío.

Número de revisiones global

A diferencia de muchos otros sistemas de control de versiones, los números de revisión de SUBVERSION se aplican a árboles enteros, no a ficheros individuales.
Cada número de revisión selecciona un árbol completo: un estado particular del repositorio tras algún cambio publicado.

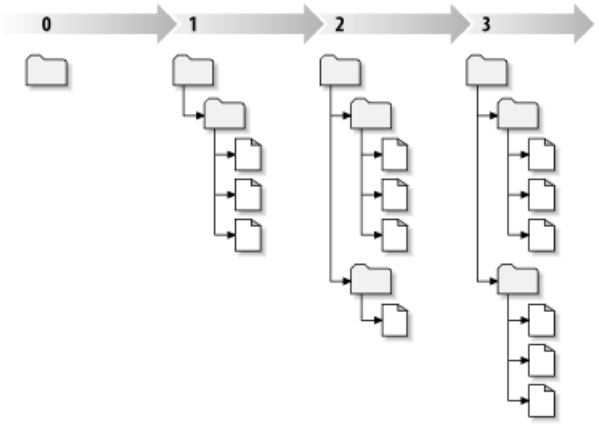

Figura 3.4: Numeración de revisiones

Otra manera de ver ésto es que la revisión N representa el estado del sistema de ficheros del repositorio tras el envío de cambios N-ésimo. Cuando en general se habla de la “revisión 3 de main.cpp”, lo que realmente se quiere decir es “main.cpp en el estado en que se encuentra en la revisión 3”.

3.4.8. Estado del repositorio

Las copias de trabajo deben tener (y tienen) un sistema para comprobar el estado de sus ficheros con respecto al estado del repositorio principal.

Para cada fichero de una copia de trabajo, SUBVERSION registra dos datos esenciales en el área administrativa *.svn*:

- revisión en la que está basado el fichero de la copia de trabajo (ésto se llama la revisión de trabajo del fichero)

- una marca de tiempo con la fecha de la última actualización del fichero desde el repositorio.

Con esta información, y comunicándose con el repositorio, SUBVERSION puede conocer el estado de un fichero determinado de la copia de trabajo, pudiendo ser uno de los siguientes:

Sin cambios y actualizado El fichero no ha sido modificado en la copia de trabajo ni se ha enviado ningún cambio sobre ese fichero al repositorio desde su revisión de trabajo. Un svn commit de ese fichero no hará nada, y un svn update del fichero tampoco hará nada.

Modificado localmente y actualizado El fichero ha sido modificado en la copia de trabajo pero no se ha enviado ningún cambio sobre ese fichero al repositorio desde su revisión base. Hay cambios locales que no han sido enviados al repositorio, por lo que un svn commit del fichero publicará con éxito sus cambios, y un svn update del fichero no hará nada.

Sin cambios y desactualizado El fichero no ha sido modificado en la copia de trabajo, pero sí en el repositorio. El fichero debería ser actualizado para sincronizarlo con la revisión pública. Un svn commit del fichero no hará nada, y un svn update del fichero introducirá los últimos cambios en su copia de trabajo.

Modificado localmente y desactualizado El fichero ha sido modificado tanto en la copia de trabajo como en el repositorio. Un svn commit del fichero fallará dando un error de "desactualizado". El fichero debe ser actualizado primero; un svn update intentará mezclar los cambios públicos con los cambios locales. Si SUBVERSION no puede combinar los cambios de manera convincente automáticamente, dejará que sea el usuario el que resuelva el conflicto.

3.4.9. Ramas en un repositorio

Definición 3.4.1. *Una rama es una línea de desarrollo que existe de forma independiente a otra, pero que comparte una historia común en algún punto temporal anterior. Se puede decir que una rama siempre nace como una copia de algo y a partir de ahí pasa a generar su propia historia.*

Figura 3.5: Ramas de desarrollo

En SUBVERSION, hemos de tener en cuenta dos detalles importantes sobre las ramas:

1. A diferencia de muchos otros sistemas de control de versiones, las ramas de SUBVERSION existen como directorios normales del sistema de archivos en el repositorio, no en una dimensión extra. Estos directorios simplemente llevan información histórica adicional.

2. SUBVERSION no tiene un concepto interno de rama. Cuando copia un directorio, el nuevo directorio sólo es una "rama" porque el usuario añade esa connotación. Puede considerar el directorio de forma diferente, o tratarlo de manera diferente, pero para SUBVERSION no es más que un directorio ordinario que simplemente fue creado como resultado de una operación de copia.

De ahí que surjan problemas muy engorrosos cuando lo que se desea es algo más complejo que crear o destruir estas ramas: cuando se desea fusionar ramas. Este tema resulta tan complejo que no vamos a entrar en él. Simplemente dejar claro que para este tipo de casos, lo más adecuado es utilizar un sistema como Git, del que hablaremos más adelante, puesto que su comportamiento está más enfocado en este sentido.

3.4.10. El repositorio

Antes de centrarnos en la instalación y administración del repositorio hemos de entender qué es realmente un repositorio y qué contiene (cómo se representa la información dentro del repositorio, qué apariencia tiene y cómo actúa un repositorio con respecto a herramientas no pertenecientes a SUBVERSION).

Entendiendo las Transacciones y Revisiones

Ya hemos hablado conceptualmente de un repositorio, una secuencia de árboles de directorios. Cada árbol es una fotografía de cómo eran los ficheros y directorios versionados en tu repositorio en un momento determinado.

Cada revisión nace como un árbol de transacciones. Cuando se envían cambios al repositorio, el programa cliente construye una transacción de SUBVERSION que copia los cambios locales (junto a cualquier cambio adicional que haya podido tener lugar desde el comienzo del proceso de envío de datos), y luego pide al repositorio que guarde ese árbol como la próxima fotografía en la secuencia. Si el envío de datos no da error, la transacción se convierte en una nueva revisión del árbol, y se le asigna un nuevo número de revisión. Si el envío de datos fallara por alguna razón, la transacción se destruye, y se informa al cliente del error.

Las actualizaciones funcionan de una manera parecida. El Cliente prepara un árbol de transacción temporal que copia el estado de la copia de trabajo. El repositorio compara entonces ese árbol de transacción con el árbol de la revisión solicitada (la más reciente salvo que se indique otra de manera específica), e informa al cliente acerca de qué cambios son necesario para convertir su copia local de trabajo en una réplica de ese árbol de revisión. Tras completarse la actualización, se borra la transacción temporal.

El uso de árboles de transacción es la única manera de hacer cambio permanentes en un repositorio de sistema de ficheros versionados. De todas maneras, es importante entender que el tiempo de vida de una transacción es completamente flexible. En el caso de actualizaciones, las transacciones con árboles temporales que se destruyen inmediatamente. En el caso de envíos al repositorio, las transacciones son transformadas en revisiones permanentes (o borradas si el envío falla). En el caso de un error, es posible que una transacción permanezca accidentalmente suelta en el repositorio (sin que afecte en realidad a nada, pero ocupando espacio).

Propiedades no versionadas

Las transacciones y las revisiones en el repositorio SUBVERSION pueden tener propiedades adjuntas. Estas propiedades son relaciones genéricas clave-valor, y generalmente se usan para guardar información acerca del árbol al que están adjuntas. Los nombres y valores de estas propiedades se guardan en el sistema de ficheros del repositorio, junto con el resto de los datos de tu árbol.

Las propiedades de revisiones y transacciones son útiles para asociar información con un árbol que no está estrictamente relacionada con los ficheros y directorios de ese árbol — el tipo de información que no es gestiona por las copias de trabajo de cliente. Por ejemplo, cuando una nueva transacción de envío es creada en el repositorio, SUBVERSION añade una propiedad a dicha transacción llamada svn:date—una marca de tiempo que representa el momento en que la transacción se creó. En el momento que el proceso de envío termina, el árbol también ha recibido una propiedad para guardar el nombre del usuario que es autor de la revisión (svn:author) y una propiedad para guardar el mensaje de informe de cambios adjunto a dicha revisión (svn:log).

Las propiedades de revisiones y transacciones son propiedades no versionada—cuando son modificadas, sus valores previos se descartan definitivamente. Así mismo, mientras los árboles de revisiones en sí son inmutables, las propiedades adjuntas de dichos árboles no lo son. Puedes añadir, borrar, y modificar propiedades de revisiones en cualquier momento más adelante. Si envías al repositorio una nueva revisión y más tarde te das cuenta de alguna información incorrecta o un error sintáctico en tu mensaje de log, puedes simplemente sustituir el valor de la propiedad svn:log con un nuevo y corregido mensaje de log.

Base de datos Berkeley

Los datos almacenados dentro de repositorios SUBVERSION, realmente se encuentran en una base de datos, más concretamente, un fichero de base de datos Berkeley. Durante la fase inicial de diseño de SUBVERSION, los desarrolladores decidieron usar una base de datos Berkeley por una serie de razones, como su licencia open-source, soporte de transacciones, ser de confianza, funcionamiento, simplicidad de su API, soporte de hilos, cursores, y más.

La base de datos Berkeley tiene un soporte real de transacciones —probablemente es su característica más poderosa. Muchos procesos que acceden a sus repositorios SUBVERSION no tienen que preocuparse por pisotear los datos de otros. El aislamiento provisto por el sistema de transacciones es tal que por cada operación dada, el código de repositorio SUBVERSION tiene una vista estática de la base de datos—no una base de datos que está constantemente cambiando de la mano de algunos otros procesos—y puede tomar decisiones basándose en esa vista. Si dicha decisión está en conflicto con lo que otro proceso esté haciendo, la operación completa como si nunca hubiera sucedido, y SUBVERSION reintenta la operación contra una nueva y actualizada (y estática) vista de la base de datos.

Otra gran característica de la base de datos Berkeley son las copias de seguridad en caliente— la habilidad para hacer una copia de seguridad del entorno de la base de datos sin que tenga que estar “”. Hablaremos sobre cómo hacer copias de seguridad de tu repositorio en “Copias de seguridad del repositorio”, pero los beneficios de ser capaz de hacer copias completas y funcionales de tus repositorios sin debería ser obvia.

La base de datos Berkeley también es un sistema de bases de datos de mucha confianza. SUBVERSION utiliza las utilidades de registro de las bases de datos Berkeley, lo que significa que la base datos primero escribe una descripción de cualquier modificación que vaya a hacer en ficheros de registros, para luego hace la propia modificación. Esto es para asegurar que si algo fuese mal, el sistema de base de datos pueda retroceder a un checkpoint—una posición en los ficheros de registro que se sabe que no están corruptas—y repetir transacciones hasta que los datos estén en un estado usable. Ver “Gestionando el espacio de almacenamiento” si quieres más información acerca de los ficheros de registro de las bases de datos de Berkeley.

Sin embargo, esta base de datos también tiene sus defectos:

- Los entornos de base de datos Berkeley no son portables.

No puedes copiar simplemente un repositorio SUBVERSION que fue creado en un sistema Unix a un sistema Windows y esperar que funcione.
A pesar de que la mayor parte del formato de base de datos Berkeley es independiente de la arquitectura, hay otros aspectos del entorno que no lo son.

- SUBVERSION usa la base de datos Berkeley de una manera que no puede funcionar en sistemas Windows 95/98 — si necesita almacenar un repositorio en una máquina Windows, utilice Windows 2000 o Windows XP.
- No se debe mantener un repositorio SUBVERSION en una unidad compartida por red. Mientras las bases de datos Berkeley prometen un comportamiento correcto en unidades compartidas por red que cumplan un grupo particular de especificaciones, casi ningún sistema de compartición conocido cumple con todas esas especificaciones.

3.4.11. Instalación, configuración y uso en Debian

Funcionamiento

Al iniciar un proyecto normalmente se instala un repositorio SUBVERSION en el servidor central, que generalmente tiene una estructura predefinida como como la que se muestra en la imagen:

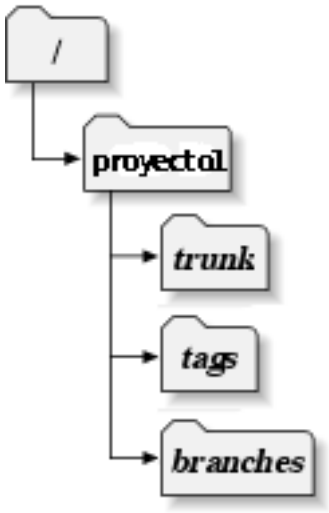

Figura 3.6: Estructura del Repositorio

Donde el repositorio proyecto1 contendría:

- *trunk*: carpeta donde reside el código fuente en desarrollo.
- *branches*: carpeta donde se realizan las pruebas sobre el código, de forma independiente para no afectar al desarrollo principal del proyecto.
- *tags*: carpeta donde están guardadas las versiones disponibles del código.

Instalación de SUBVERSION

La instalación[1] en sistemas tipo Debian es muy sencilla:

```
sudo apt-get install subversion
sudo apt-get install subversion-tools
```

Configuración

En principio, el servidor svn como tal no necesita configuración adicional, pero si se pretende utilizar el repositorio de forma remota se ha de usar el servidor svnserve.

Lo adecuado es crear un nuevo grupo para SUBVERSION y luego añadir algún usuario distinto de *root* que pueda ejecutar el servidor SVNSERVE:

[1] Lógicamente se ha de instalar tanto en el servidor como en los clientes.

```
sudo groupadd subversion
sudo addgroup usuario subversion
```

Para arrancar el demonio se ejecuta svnserve:

```
svnserve -d -r /home/demo/repository
```

SVNSERVE es un servidor independiente, ejecutable como proceso demonio o invocable por SSH[2].

NOTA: La opción d es para que arranque a modo de demonio y la opción r es para que funcionen los repositorios que pudieran estar instalados por debajo del directorio *repositorio/*.

Para su configuración se utiliza el archivo */repositorio/conf/svnserve.conf*, pero como avisa en el encabezamiento de dicho archivo si se van a usar URLs de los tipos *http://* y *file:///*, la configuración que se incluya en este archivo es totalmente ignorada.

Creación de nuestro primer repositorio en el servidor

Una vez instalado el svn podemos rápidamente crear el repositorio en algún directorio de nuestro servidor. Para ello basta con ejecutar:

```
svnadmin create /path/repositorio
```

Creamos un directorio temporal y entramos en él:

```
mkdir tmpdir
cd tmpdir
```

Dentro debería ir toda la estructura de directorios anteriormente comentada, así que creamos los directorios y subdirectorios:

```
mkdir proyecto1
mkdir proyecto1/trunk
mkdir proyecto1/tags
mkdir proyecto1/branches
```

Para añadir los proyectos, si se tiene un proyecto empezado, se reorganizaría en las carpetas arriba comentadas, trunk, tags, branches. Colocando todo el código de nuestro proyecto en la carpeta que le corresponda, normalmente en *trunk* o en *branches* en caso de que fueran pruebas ajenas al código real.

Una vez creadas las carpetas y dispuesto el código, hay que importar desde el repositorio dicha estructura:

```
svn import . file:///path/hasta/repositorio/proyecto1 \
    -message 'Creando el primer repositorio'
```

NOTA: El . en el comando anterior indica que la fuente del código inicial para el repositorio está en el mismo directorio donde nos encontramos, es decir, que se halla en */tmpdir*.

En general, los URLs de SUBVERSION utilizan la sintaxis estándar, permitiendo la especificación de nombres de servidores y números de puertos como parte del URL.

[2] Otra alternativa para hacer que el repositorio esté disponible remotamente

Una vez importado el código, nos podemos deshacer del directorio temporal, o si se prefiere, esperar a que todo funcione correctamente para evitar pérdidas engorrosas.

```
cd ..
rm -rf tmpdir/
```

Activar el módulo de Apache de SUBVERSION

Para que se puedan mostrar los repositorios a través de la web se ha de activar el servidor apache, para lo cual se ha de instalar la librería correspondiente:

```
sudo apt-get install libapache2-svn
```

Normalmente tras la instalación de esta librería lo lógico es que se active el módulo requerido, pero ejecutaremos el comando siguiente para asegurarnos:

```
sudo a2enmod dav_svn
```

Configuración del módulo de SUBVERSION

Para configurar este módulo se ha de editar el archivo *dav_svn.conf* que se encuentra en el directorio */etc/apache2/*.

```
<Location /svn>

DAV svn

# SVNPath, permite acceder via apache a la
# direccion: dominio/svn directamente

SVNPath /home/Usuario/svn

# SVNParentPath, permite acceder a directorios
# superiores al repositorio, si y solo si,
# apache tiene permisos de lectura sobre dichos
# directorios.

# SVNParentPath /home/Usuario/svn

# No pueden estar activados estas dos opciones
# (SVNParentPath y SVNPath) simultaneamente.

AuthType Basic
AuthName "Repositorio \textsc{Subversion} del proyecto"
AuthUserFile /etc/apache2/dav_svn.passwd

<LimitExcept GET PROPFIND OPTIONS REPORT>

# Require valid-user
```

```
</LimitExcept>
</Location>
      CustomLog /var/log/apache2/svn/access.log combined
      ErrorLog /var/log/apache2/svn/error.log
```

Tras modificar este archivo, siempre, se ha de reiniciar el servidor web:

```
sudo apache2ctl restart
```

Para comprobar si se puede ver el repositorio a través de la web, sólo necesitamos utilizar un navegador cualquiera:

```
firefox http://dominio:puerto/repositorio/
```

Siendo *dominio* la IP real del servidor, *repositorio* el nombre del repositorio creado y no siendo necesario incluir *:puerto* (puerto de acceso), puesto que por defecto se toma el puerto 80.

Si no hay ningún problema y se puede navegar en el repositorio, ya está listo para funcionar.

Se pueden incluir scripts y mejoras al nuestro repositorio para que se visualice de forma más estética desde la web, utilizando por ejemplo WebSVN, aunque este tema excede lo que abarca este breve tutorial.

Cantidad de repositorios y organización

Se pueden tener tantos repositorios como se deseen en un mismo servidor, y también se pueden tener tantos proyectos en un mismo repositorio como se necesiten, en realidad es cuestión de gustos elegir una u otra forma de administrar los repositorios.

3.4.12. Utilización y manejo

- Crear un nuevo repositorio:

```
svnadmin create repositorio
```

- Hacer copias de seguridad del repositorio:

```
svnadmin dump /paht/repositorio/proyecto1 | gzip -9 > \
        dump_svn_proyectoA.gz
```

  NOTA: Resultaría conveniente incluir la órden en el CRON, para que se hagan las copias de seguridad con cierta periodicidad de forma automática.

- Si es necesario borrar un repositorio por alguna razón, basta con borrar el directorio en el que se ubica en el disco duro del servidor.

```
sudo rm -r /paht/hasta/repositorio/proyecto1
```

  NOTA: Este paso no se puede deshacer.

3.4.13. SUBVERSION en la web

Últimamente, vienen apareciendo servicios interesantes basados en la Web para probar SUBVERSION. Algunos ofrecen cuentas gratuitas, limitadas en su funcionalidad, y otros están destinados a un uso fuerte del sistema.

Entre ellos podemos destacar los siguientes:

Beanstalk (www.beanstalkapp.com)
: Servicio nuevo y con buena interfaz, realmente muy fácil de usar. Es gratuito con hasta 20 MB de almacenamiento. Se destaca su excelente soporte. Es ideal para los principiante.

Roundhaus (www.roundhaus.com)
: Aunque la cuenta gratuita tiene sus limitaciones, se destaca por ser uno de los pocos que ofrecen un servicio de integración continua, lo cual lo convierte en algo más que un simple repositorio en la Web.

Google Code (http://code.google.com)
: Dentro del amplio abanico de servicios gratuitos de Google, se destaca Google Code, que ofrece hosting gratuito ilimitado con acceso, claro, a SUBVERSION.

En el ámbito español, podemos incluir también las forjas de desarrollo, como por ejemplo:

Forja de RedIris (https://forja.rediris.es/)
: La Forja de la Comunidad RedIRIS ofrece la infraestructura necesaria para que los generadores de conocimiento libre puedan comunicarse entre sí y coordinarse remotamente en el desarrollo del proyecto. La máquina que alberga la Forja está instalada en las dependencias del Centro Informático Científico de Andalucía (CICA), el cual colabora en este proyecto.

Forja de OSLUCA (www.uca.es/softwarelibre/forja)
: Con funcionalidades especiales para Proyectos de Fin de Carrera.

Capítulo 4

El servidor en Sistemas Distribuidos

4.1. Introducción a los Sistemas de Control de Versiones Distribuidos

Los DVCS son herramientas de control de versiones que toman un enfoque punto a punto (*peer-to-peer*), al contrario de los centralizados que toman un enfoque *cliente-servidor*. Con ellos solo se mantiene una copia local del repositorio, pero cada equipo se convierte en un **repositorio** del resto de usuarios. Esto permite una fácil bifurcación en ramas y un manejo más flexible del repositorio.

En este sistema existen pocos comandos que no trabajen directamente con el propio disco duro, lo que lo convierte en una herramienta más rápida que un **Sistema de Control de Versiones Centralizado**. Cuando hayamos decidido que estamos listos, podemos enviar los cambios a los repositorios de los otros usuarios. Otra de las ventajas de trabajar de forma local es que puedes trabajar sin conexión a la red.

En este sistema es importante la diferencia entre ramas de desarrollo (destinadas a pruebas, inestables, etc...) y la rama principal (trunk/master), en principio la actualización del repositorio se llevará a cabo de la rama principal y no del resto de ramas, aunque últimas sí que pueden actualizarse en repositorios externos.

4.2. El enfoque distribuido

4.2.1. Diferencias entre DCVS y CVS

Las principales diferencias entre un sistema centralizado y uno distribuido son las siguientes, algunas ya señaladas en la introducción, son las siguientes:

1. No existe una copia de referencia del código, solo copias de trabajo.
2. Las operaciones suelen ser más rápidas al no tener que comunicarse con un servidor central.
3. Cada copia de trabajo es un tipo de respaldo del código base.
4. No hay que hacer `update` antes del `commit`, puesto que se trabaja sobre una copia local.
5. Se eliminan los problemas de latencia de la red.
6. La creación y fusión de ramas es más fácil, porque cada desarrollador tiene su propia rama.

Las principales desventajas que encontramos son las siguientes:

1. Todavía se necesita un sistema de backup. No hay que fiarse de que el backup reside en otro usuario, ya que este puede no admitirte más, o estar inactivo mientras que yo tengo cambios hechos, por lo que todavía será necesario un servidor central donde realizar los backups.

2. Realmente no hay una *última versión*. Si no hay un repositorio central no hay manera de saber cuál es la última versión estable del producto.

3. Realmente no hay números de versión. Cada repositorio tiene sus propios números de revisión dependiendo de los cambios. En lugar de eso, la gente pide la última versión del guid (número de versión) concreto, aunque cabe la posibilidad de etiquetar cada versión.

Encontramos dos nuevos comandos principales que no trabajan localmente:

- `push`: envía cambios a otro repositorio (por supuesto, requiere permisos).
- `pull`: obtiene los cambios de otro/s repositorios.

4.2.2. Cómo funciona

Como se ha dicho cada desarrollador tiene su repositorio local: si un usuario A realiza un cambio, lo hace localmente y cuando crea oportuno puede compartirlo con el resto de usuarios.
Pero existe una duda, ¿cuál es la versión principal? La respuesta es sencilla: existe una rama principal en la cual todos los usuarios registran los cambios realizados obteniendo de esta forma la versión final.

Para más detalles explicaremos esto de forma gráfica como se muestra en la Figura 4.1.

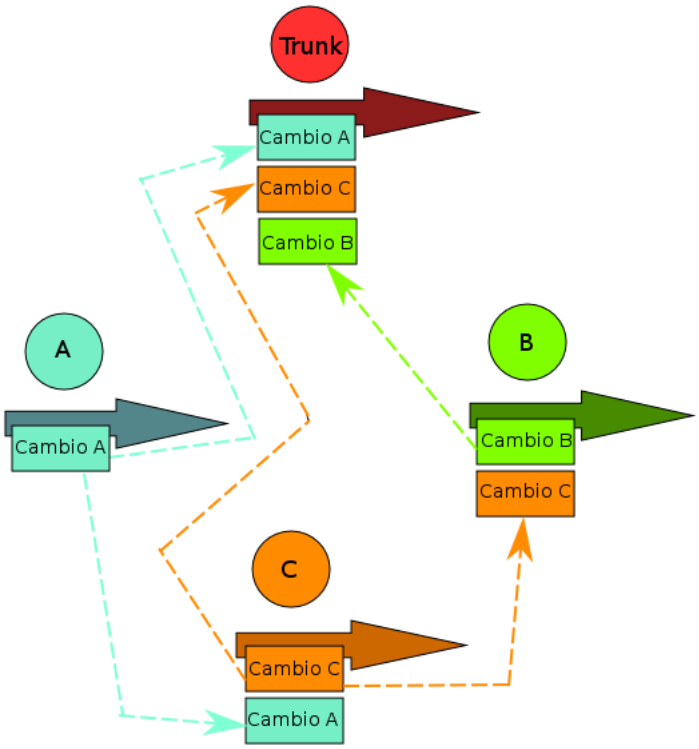

Figura 4.1: Funcionamiento de un DCVS

En el gráfico podemos ver la rama principal (*trunk*) y varios usuarios (A, B, y C). Cada usuario tiene su propio repositorio con ciertas ramas de desarrollo. Existe una rama principal que lleva las versiones oficiales y sin fallos. Cada usuario guarda su copia local, pero además envía y recibe desde y a otros repositorios ciertas ramas de desarrollo y por supuesto la principal que es la que contiene las versiones estables del proyecto.

4.2.3. Diferentes DCVS

Entre los más populares encontramos:

- ⋆ Git
- ⋆ Mercurial
- ⋆ Baazar
- ⋆ Darcs
- ⋆ Monotone

Procedemos a explicar las bases de Git y Mercurial, al ser los más utilizados actualmente.

4.3. GIT

GIT es uno de los sistemas de control de versiones distribuidos más usados. Fue creado por Linus Torvalds[1] y buscaba un sistema que cumpliera 4 requisitos básicos:

- No parecido a CVS
- Distribuido
- Seguridad frente a corrupción, accidental o intencionada
- Gran rendimiento en las operaciones

GIT está escrito en C y en gran parte fue construido para trabajar en el kernel de Linux, lo que quiere decir que desde el primer día ha tenido que mover de manera efectiva repositorios de gran tamaño.

4.3.1. GIT es distribuido

Cada usuario tiene una copia completa del servidor principal, y cualquiera de ellas podría ser recuperada para reemplazarlo en caso de caída o corrupción. Básicamente, no hay un punto de fallo único con git a no ser que haya un punto único.

4.3.2. Ramas locales sin coste

Posiblemente la razón más fuerte a favor de GIT que realmente lo hace destacar de casi cualquier otro SCV es su modelo de ramas. GIT permite tener múltiples ramas locales independientes entre sí y se tarda segundos en crear, fusionar o borrar estas líneas de desarrollo.

Se pueden hacer cosas como:

- Crear una rama para probar una idea, volver al punto desde el cual bifurcaste y volver a donde estabas experimentando y fusionarlo.
- Tener una rama que siempre contiene sólo lo que va a producción, otra en la que acumulas el trabajo para testear y varias ramas más pequeñas para el trabajo diario.
- Crear una rama en la que experimentar, darte cuenta de que no va a ninguna parte y borrarla, abandonando todo el trabajo (sin que nadie más lo vea).

Es importante que, cuando un usuario entrega sus cambios a un repositorio remoto, no tiene que subir todas sus ramas: se puede compartir sólo una de tus ramas y no todas. De esta forma la gente tiende a sentirse libre para probar nuevas ideas sin preocuparse de tener un plan de cómo y cuándo van a mezclar sus cambios o compartirlos con otros.

4.3.3. GIT es local, rápido y pequeño

Al ser local, trabaja de forma muy rápida y también permite trabajar sin conexión. Puedes descargar la información del servidor y trabajar con ella de forma local.

Existen pocos comandos que accedan al servidor y los repositorios ocupan muy poco espacio.

4.3.4. El área de montaje

GIT tiene lo que se denomina “área de montaje” o “índice”, que es un área intermedia donde puedes configurar el aspecto que tendrá tu entrega antes de hacer el `commit`. Esto también permite preparar sólo fragmentos de archivos que han sido modificados. Por ejemplo, montas para entregar sólo los cambios al principio de un archivo que has estado modificando, pero no los cambios del final. Por supuesto, GIT también hace que sea fácil ignorar esta funcionalidad en caso de que no queramos tanto nivel de control (simplemente se añade ’-a’ a la órden `commit`).

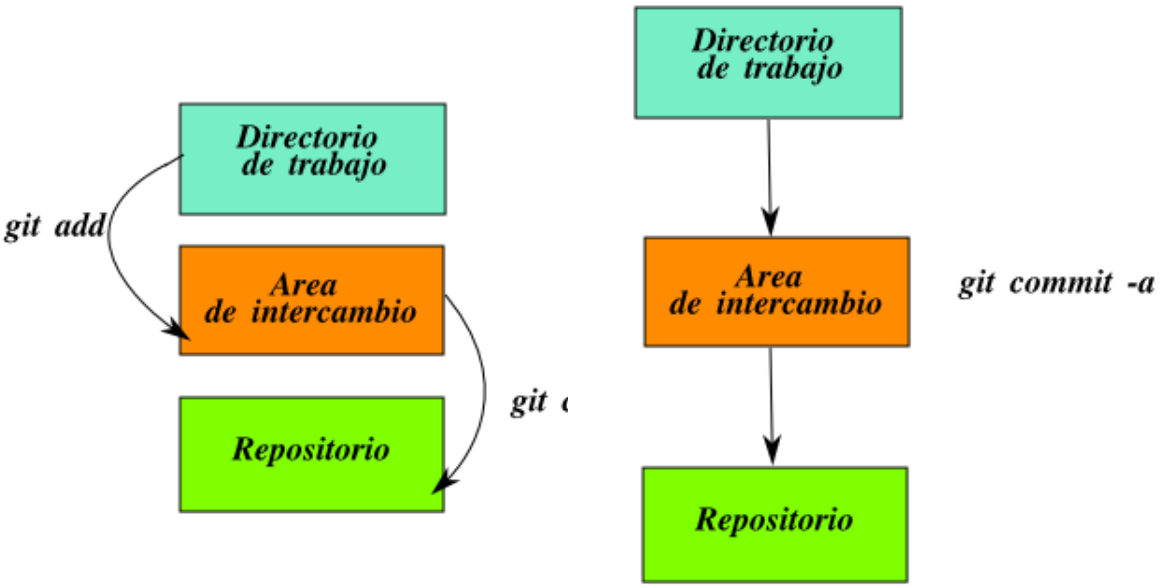

Figura 4.2: Formas de realizar un *commit*

4.3.5. Diferentes flujos de trabajo

Se puede implementar fácilmente casi cualquier flujo de trabajo que se nos ocurra.

- **Estilo SUBVERSION**
  Forma de trabajo muy común entre aquellos que se pasan de un sistema no distribuido es un flujo de trabajo centralizado. GIT no permite hacer subir los cambios al servidor si alguien lo ha hecho desde la última vez que nos bajamos los últimos cambios, de forma que un modelo centralizado donde todos los desarrolladores entregan al mismo servidor.

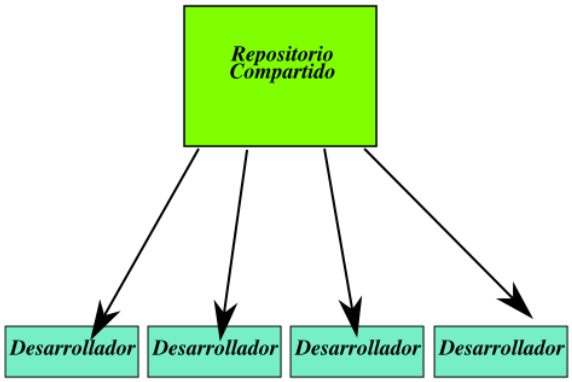

Figura 4.3: Estilo SUBVERSION

- **Estilo Responsable Integración**
  Existe un Responsable de Integración, una persona que entrega al repositorio final, y después un número de desarrolladores se copian ese repositorio y hacen sus cambios, luego piden al integrador que integre sus cambios.

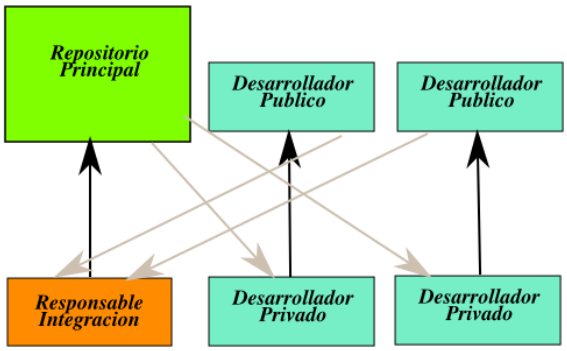

Figura 4.4: Responsable integración

- **Estilo Dictador y Tenientes**
  Similar a como lo hacen en el kernel de Linux, donde hay gente que está a cargo de un subsistema específico del proyecto (los tenientes) e integran todos los cambios que tienen que ver con ese subsistema. Después otro integrador (el dictador) puede recoger los cambios únicamente de sus tenientes y subirlos al repositorio principal, el cual todo el mundo vuelve a clonar.

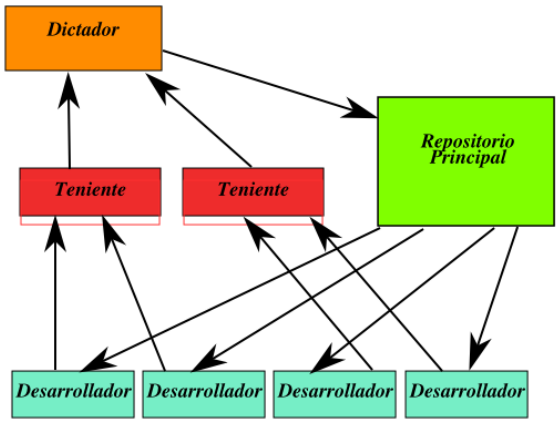

Figura 4.5: Dictador y Tenientes

4.3.6. Modelo de objetos

Toda la información necesaria para representar el historial de un proyecto se guarda en los archivos de referencia mediante un nombre de objetos de 40 dígitos. Se calcula tomando el hash SHA1 de los contenidos del objeto por lo que es muy dificil encontrar dos objetos iguales. Esto tiene una serie de ventajas:

- Se puede determinar si dos objetos son o no el mismo rapidamente mirando el nombre.
- El mismo contenido en dos repositorios será alamcenado econ el mismo nombre.
- Se pueden detectar errores facilmente con el nombre del objeto.

Los objetos se componen de: un tamaño, contenido (depende del tipo objeto) y un tipo, exites 4 tipos:

Blob: Se usa para almacenar archivo de datos. Es unfragmento de datos binarios, no posee atributos ni nombres de archivo.

Tree: Básicamente es como un directorio. Hace referencia a otros tree y blobs. Este objeto contiene una lista de entradas con los nombres y los nombres SHA1. donde cada blob es un archivo y cada tree un subdirectorio.

Commit: Contiene meta-información como el punto en el tiempo, autor, puntero al `commit` anterior. Conecta el estado físico de un árbol con una descripción de como se llega allí y porque. Un árbol está definido por:

- Un objeto tree.
- Un padre, que es un nombre SHA1 con los `commits` anteriores.
- Responsable del cambio con su fecha.
- El usario que hace el `commit` actual.
- Un comentario.

Tag: Se utiliza principalmente para etiquetar commmits específicos.

Casi todo en GIT consiste en manejar estos cuatro objetos, se trata de una especie de pequeño sistema de ficheros.

4.3.7. Directorio GIT y directorio de trabajo

El directorio GIT es el directorio que almacena todos los meta-datos de la información del proyecto, los objetos y los punteros a las diferentes ramas entre otras cosas. Solo existe un directorio GIT por proyecto a diferencia de SUBVERSION donde existe uno por directorio. Este directorio es *.git* localizado en la raíz del proyecto por defecto.
Los principales archivos son los siguientes:

- ⋆ `HEAD`: Puntero a la rama actual.
- ⋆ `config`: Fichero de configuración.
- ⋆ `description`: Descripción del proyecto.
- ⋆ `hooks/`: Acciones pre/post.
- ⋆ `index`: Fichero principal.
- ⋆ `logs/`: Historial de las ramas.
- ⋆ `objects/`: Objetos del proyecto.
- ⋆ `refs/`: Punteros a las ramas.

El directorio de trabajo es el directorio donde se encuentras los archivos con los que se esta trabajando. Es el lugar donde se modifican estos archivos hasta el siguiente `commit`.

4.3.8. Archivo Index

Es un área entre el directorio de trabajo y el repositorio. Se usa para crear los cambios realizados en un `commit`, cuando se hace un `commit` lo que se modifica se hace sobre este archivo y no sobre el directorio de trabajo. Si lo elimina por completo no se pierde toda la información siempre que se tenga el nombre del árbol (tree) que describe.

4.3.9. Uso de GIT

Se rige por un comportamiento básico de "update-modify-commit". A continuación se incluye una lista de las instrucciones más comunes con su equivalente en SVN:

Equivalencias

ÓRDEN EN GIT	EQUIVALENTE EN SUBVERSION
`git clone <repositorio>`	`svn checkout <repositorio>`
`git init (desde el directorio)`	`svn admin create <repositorio>`
`git add <fichero>`	`svn add <fichero>`
`git rm <fichero>`	`svn rm <fichero>`
`git checkout <ruta>`	`svn revert <ruta>`
`git pull`	`svn update`
`git commit -a`	`svn commit`
`git diff`	`svn diff | less`
`git status`	`svn status`
`git tag/git branch`	`svn copy <origen> <destino>`

Tabla 4.1: Equivalencias entre GIT-SUBVERSION

4.3.10. Que es GibHub

Gibhub es más que un sitio de alojamiento: se ha convertido en una red social de código, donde encontramos a otros desarrolladores y donde es muy fácil bifurcar y contribuir, creándose una comunidad ven torno a GIT y los que proyectos en los que la gente lo usa.

4.3.11. Primeros pasos

Supongamos que tenemos GIT instalado y un repositorio al algún sitio remoto. Una vez hecho esto, tendremos que hacer un `checkout` cuyo equivalente en git es `clone`. Vamos a usar el repositorio público de proyectos para este ejemplo.

En primer lugar se muestran los pasos para crear un nuevo repositorio en GibHub.Para ello nos damos de alta en GibHub y creamos un nuevo repositorio en la web. Los siguientes pasos son para crear un nuevo repositorio local y hacer nuestro primer `commit` (local) y `push` (remoto):

```
mkdir Trabajo-ASO
cd Trabajo-ASO
git init
git add .
git commit -a -m 'Primer commit'
git remote add origin git@github.com:solidge0/Trabajo-ASO.git
git push origin master
```

Ooopps, error, no tenemos llave pública, para el caso de GIT se puede emplear la seguridad de un certificado RSA el cual podemos crear desde el directorio home de nuestro usuario:

```
cd .ssh
ssh-keygen -t rsa
```

Ahora en el archivo *id_rsa.pub* tenemos nuestra llave pública, así que la copiamos en GitHub para poder subir nuestros archivos.

Ya tenemos un nuevo repositorio, pero ¿y si este existía, no lo tenemos y queremos bajarlo? Fácil:

```
git clone git://github.com/solidge0/Trabajo-ASO.git
```

Y con esto creamos un directorio con los archivos del repositorio. Ahora podemos hacer:

```
git status
```

Y se nos mostrará el estado actual. Y siempre que hagamos un `commit` este será local, para actualizar la rama principal al servidor haremos:

```
git push git@github.com:solidge0/Trabajo-ASO.git
```

El URL que hemos usado es diferente: mientras que antes habíamos usado un URL genérico, que no permite cambiar lo que hay, habrá que usar el anterior para subir los cambios[2].

Ahora ya podemos bajar los cambios que se realizen mediante:

[2]Para ejecutar esta orden tendremos que tener permiso.

```
git pull
```

4.3.12. Branch y merge

Como ya se comentó anteriormente GIT puede tener múltiples ramas de desarrollo. Además de la principal podemos crear otras ramas, para ello:

```
git branch prueba
```

Ahora ejecutamos el siguiente comando veremos una lista de ramas, donde aparece la principal por defecto (*master*) y la nueva creada, la rama actual activa :

```
git branch
```

Para cambiar a la nuevarama creada:

```
git checkout prueba
```

Ahora si queremos fusionar ambas ramas para combinar los cambio realizados en cada una, cambiamos por ejemplo a la rama master:

```
git checkout master
git merge prueba
```

Y si los cambios no entran en conflicto ambas se fusionan. Si existen conflictos hay que solucionarlos manualmente, podemos verlos con:

```
git diff
```

Para borrar una rama que ya no queramos usa:

```
git -d prueba
```

Este comando se asegura de que los cambios de la rama que se borra se encuentran en la rama actual. Si usamos el modificador -D no tendrá en cuenta que los cambios estén en la rama actual.

También podemos deshacer los cambios realizados con una fusión (`merge`):

```
git reset --hard HEAD
```

Pero si ya se ha realizado el `commit`:

```
git reset --hard ORIG\_HEAD
```

Aunque esto último puede crear problemas si esta rama se ha fusionado otra vez con alguna otra rama.

4.4. MERCURIAL

MERCURIAL es otro sistema de control de versiones muy extendido, el creador y desarrollador principal de MERCURIAL es Matt Mackall. Está implementado principalmente haciendo uso del lenguaje de programación Python, pero incluye una implementación binaria de diff escrita en C.

MERCURIAL fue escrito originalmente para funcionar sobre Linux. Ha sido adaptado para Windows, Mac OS X y la mayoría de otros sistemas tipo Unix. Los defensores de MERCURIAL afirman que tiene la facilidad de SUBVERSION y la potencia de GIT. Como este documento no pretende ser una comparativa y dada la similitud con GIT y SUBVERSION se hará un repaso rápido de este sistema de control de versiones. Podemos ver la similitud con GIT en sus órdenes[3].

Para iniciar un directorio:

```
hg init
```

Añadir al repositorio lo que hubiera en ese directorio:

```
hg add
```

MERCURIAL dispone de los comandos habituales en un SCV tradicional:

```
add
commit (ci)
update (up)
status (st)
```

Pero al ser mercurial un sistema de control de versiones distribuido, estas operaciones ocurren sobre un repositorio local. Para trabajar con un repositorio ajeno dispone de comandos similares a GIT:

```
clone
pull
push
```

[3]Todas las operaciones de MERCURIAL se invocan como opciones dadas a su programa motor, hg, que hace referencia al símbolo químico del mercurio.

Bibliografía

[1] *Subversion, instalación, configuración y uso en Debian*. http://alufis35.uv.es/~laura/spip/spip.php?article183.

[2] C. Michael Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. *Libro Rojo de SVN* (bajo licencia creative commons). http://svnbook.red-bean.com/nightly/es/index.html.

[3] Roberto García Carvajal. *Subversion*. http://aulavirtual.uca.es/moodle/file.php/7587/svn/seminario_subversion.pdf.

[4] Equipo de Desarrollo de GIT. *Documentación oficial*. http://git-scm.com/documentation.

[5] Antonio García Domíngez. *Apuntes-Operaciones Básicas*. http://gitorious.org/projects/curso-git-osluca/repos/mainline (sin servicio).

[6] Ismael Olea e Ignacio Arenaza. *Tutorial de uso de* CVS. http://es.wikibooks.org/wiki/Tutorial_de_uso_de_CVS.

[7] Wikipedia. *Control de Versiones*. http://es.wikipedia.org/wiki/Control_de_versiones.

GNU Free Documentation License

Version 1.3, 3 November 2008

<http://fsf.org/>

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.
The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.
A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".
Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.
The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.
The "**publisher**" means any person or entity that distributes copies of the Document to the public.
A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.
The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled “History”, Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled “History” in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the “History” section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.
You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.
However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.
Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.
"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.
"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.
An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.
The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.