Sistemas Operativos

Trabajo Práctico Especial 2

Implementación de un sistema de versionado concurrente

Alumnos:

Alderete, Facundo - 51063

Buireo, Juan Martín - 51061

Perez Cuñarro, Javier - 49729

ITBA - Cuatrimestre 2 - 2012

Índice

1. Introducción

Este trabajo práctico resultó en el diseño y elaboración de un sistema de versionado de archivos (estilo **SVN** o **GIT**), que ofrezca la posibilidad de crear un proyecto y almacenar varias versiones de los archivos que lo componen. Siempre que el usuario lo desee, se guardará una nueva versión de los archivos, pudiendo volver a versiones anteriores en caso de que el cambio no sea satisfactorio.

La codificación, hecha completamente en lenguaje **ANSI C**, está preparada para ser embebida en el sistema operativo **Minix** (versión 3), de modo de que el usuario pueda acceder a los servicios de nuestra aplicación desde cualquier directorio dentro del *filesystem* de **Minix**.

2. Diseño e implementación

2.1. Almacenamiento de datos

Debido a que la funcionalidad básica de nuestra aplicación consiste en almacenar datos, utilizamos archivos para este fin. Con esta metodología, y utilizando un formato de escritura propio (detallado a continuación), logramos guardar el INODO del archivo en cuestión, su número de versión y su *path* dentro del sistema de versionado del servidor. Gracias a esta metodología, nos ahorramos el hecho de tener un servidor "corriendo de fondo" que consume recursos de memoria.

Ejemplo: `12312-1-include/JUEGO.H`

Cada línea dentro del archivo `".cvs"` es del estilo del ejemplo anterior, y almacena los datos de todos los archivos dentro de una carpeta. En el caso del ejemplo, se identifica al archivo con inodo número 12312, versión 1, con *path* `"include/JUEGO.H"`. Por otro lado, en este mismo archivo se almacena la última fecha en que algún cliente hizo un `commit`.

En el servidor se cuenta con una carpeta por cada archivo inicial que se crea. Así, si alguien agrega por primera vez el archivo `JUEGO.H` se creará una carpeta 12312. De esta manera, cuando haya alguna versión nueva del archivo `JUEGO.H` esta misma se almacenará en dicha carpeta por ser "el mismo archivo" pero en una versión más reciente. Con esto nos aseguramos que el archivo se identifica por su inodo (que es único para cada archivo) y no por su nombre, permitiendo volver un archivo a versiones anteriores.

Por otra parte, ya que la identificación de archivos se hace mediante el inodo, y teniendo en cuenta que los archivos en el servidor y en el cliente, a pesar de ser el mismo en contenido tendrán inodos distintos, necesitamos establecer un mapa que relacione estos archivos. Esto se llevó a cabo relacionando los archivos mediante su nombre, entonces se pudo establecer la conexión entre estos inodos, el del archivo en el cliente con su contraparte en el servidor. Cabe aclarar que estos archivos a modo de "mapa" existen de a uno por cliente.

Ejemplo: `1-12312-45645`

En el ejemplo se observa que el primer parámetro es la versión del archivo, el segundo se corresponde con el inodo con el cual se corresponde dicho archivo en el servidor y el tercero es el inodo propio del archivo del lado del cliente.

El cliente también cuenta con un archivo (llamado `.cvs`) donde se almacenan todos los *paths* de los archivos que pertenecen al versionado. Esto sirve para a la hora de ejecutar cambios solo hacerlo sobre los archivos necesarios y no incluir otros sin versionado.

Tanto en el `.cvs` como en el `.map` del cliente se almacena la última fecha donde se introdujo alguna modificación (un `checkout`, un `update` o un `commit`). Esto sirve para poder saber si hay versiones nuevas antes de realizar un `commit`.

3. Funciones relevantes

3.1. Checkout

Se encarga de crear una nueva copia cliente del proyecto. Esto significa que esta copia podrá estar sincronizada con los cambios que hagan las demás copias sobre el proyecto y podrá introducir sus propias modificaciones.

En primer lugar se verifica que no exista otro proyecto sincronizado en la misma carpeta donde se intenta hacer el `checkout`, ya que esto podría introducir inconsistencias y errores. Esta verificación se lleva a cabo preguntando por la existencia de otro archivo con el nombre `.cvs`, ya que es este el encargado de señalar la existencia de un proyecto en sincronización con una carpeta.

En segundo lugar se necesita de la existencia del archivo `.cvs` del servidor, ubicado dentro de la carpeta `.Server`. Es ésta la encargada de sincronizar los parámetros del cliente con los del servidor. El último paso antes de sincronizar la nueva copia, es crear el directorio raíz a partir del cual se efectuará la misma. Éste es simplemente una nueva carpeta con el nombre del proyecto como identificación, y de la cual descienden todos los archivos y directorios del proyecto.

Finalmente, el algoritmo dentro de la función `checkout` se encarga de recorrer el archivo `.cvs` del server, levantando todos los nombres de archivos, que se encuentran expresados en el formato que se explicó anteriormente, y va duplicando el *path* que se indica en cada una de estas líneas, tomando como directorio raíz a la carpeta en la cual se invocó la función del `checkout`.

3.2. Add

En primer lugar, la función verifica que el directorio actual esté sincronizado con algún proyecto. Luego se procede a analizar los parámetro con los que se

invocó a la función, almacenados bajo la forma del `argv` de **C**. Se verifica que el archivo que se está intentando agregar al repositorio realmente exista en el directorio actual, ya que si esto no ocurre se produciría una inconsistencia.

Seguidamente se verificará que el archivo a agregar no forme parte actualmente del sistema de archivos del repositorio, ya que de ser así no tendría sentido agregarlo. Esto se lleva a cabo revisando el archivo que contiene el mapa de relaciones entre los inodos de un archivo en el servidor y su contraparte en el cliente, que llamamos "mapa".

Si todas las verificaciones son exitosas, se deja una marca de los cambios hechos en el repositorio (en este caso el agregado de un archivo nuevo) y la función finaliza.

3.3. Delete

Esta funcionalidad es la encargada de borrar un archivo del repositorio. Comienza verificando que el directorio actual esté efectivamente sincronizado con algún proyecto, hecho necesario para su funcionamiento. Luego se verifica que el archivo a borrar esté dado de alta en el repositorio y en el directorio de donde se intenta borrarlo.

A continuación, de ser válidas todas las verificaciones anteriores, el archivo será dado de baja lógicamente. Esto quiere decir que se eliminará su entrada en el archivo `.cvs`, pero no se lo eliminará del repositorio físicamente. Esto fue diseñado en caso de que el cliente quiera recuperar un archivo, ya que el mismo no dejará de existir del lado del servidor. El archivo eliminado simplemente no aparecerá más en la lista de los sincronizados, entonces no se lo tendrá en cuenta en las próximas llamadas a `update` o `checkout`.

3.4. Commit

Se utiliza para enviar cambios al repositorio desde una copia de trabajo local. Un `commit` puede agregar, borrar o cambiar el contenido de uno o más archivos. También puede añadir o quitar directorios en los cuales hayan archivos.

En primer lugar, se revisa que el cliente haya trabajado sobre la versión actual del repositorio. Para ello, se verifica que la fecha de los archivos del repositorio en el servidor no sea más reciente que la fecha de los archivos locales. Cabe destacar que este sistema de versionado de archivos no realiza *merges* automáticos de archivos, dejando que el usuario se encargue de esta tarea.

Asimismo, se realiza un bloqueo sobre los archivos del servidor de tal forma que, si dos o más usuario quiere realizar un `commit` al mismo tiempo, sólo se lo permita al primero que envíe el pedido. Esto asegura que no se creen inconsistencias en los clientes involucrados con respecto a la copia del servidor.

3.5. Diff

La utilidad `diff` sirve para comparar las versiones de los archivos locales del cliente con los archivos actuales del servidor. Para ello, se listan los números de versión y las fechas de modificación de todos los archivos, presentando primero los del cliente y luego los del servidor. Esto resulta útil para resolver conflictos a la hora de hacer un commit, aunque siempre es conveniente recurrir a comandos propios de **Unix** como `merge`.

3.6. Versions

Recibe como parámetro el nombre de un archivo, y se encarga de devolver la versión actual en la que se encuentra el archivo, junto con una lista de las diferentes versiones por las cuales fue pasando previamente y los nombres de archivo correspondientes a las mismas.

3.7. GetVersion

Recibe como parámetro el nombre de un archivo y un número de versión. Su funcionalidad consiste en hacer que el número de versión que se encuentra en el parámetro pase a ser la versión activa. Esto tiene como efecto que se haga una copia de la versión pedida y que esta copia pase a ser la última versión del archivo, teniendo el siguiente número en la lista de versiones.

En caso de que se pida un número de versión erróneo, la función no tendrá una salida exitosa, e informará al usuario del error cometido. A su vez, se lleva a cabo el bloqueo de los archivos del servidor, ya que al igual que con el `commit` no es deseable que dos o más clientes modifiquen parte del servidor a la misma vez.

3.8. Update

Pone al día los archivos del repositorio local con respecto a la revisión HEAD del servidor, es decir, la última versión de los mismos.

Esta función recorre el índice con todos los archivos locales del cvs. Para cada archivo, se reemplaza el contenido del archivo local con el contenido presente en el servidor.

3.9. Rename

Recibe como parámetro el *path* del archivo cuyo nombre se quiere modificar, junto con el nuevo nombre deseado. El pedido del *path* está relacionado con la posibilidad de haber dos o más archivos con mismo nombre pero carpeta distinta.

3.10. DeletedFiles

Se encarga de mostrar al usuario la lista de los archivos que fueron eliminados del repositorio. Recordemos que un archivo "eliminado" sigue existiendo dentro del servidor, aunque haya sufrido una baja lógica dentro de la lista de archivos sincronizados en el repositorio. Esto ayudará al usuario a identificar los archivos borrados recientemente y a así poder utilizar la función de recuperación.

3.11. GetDeletedFile

Se especifica un *path* que corresponde al archivo a restaurar. Este archivo debe estar necesariamente dado de baja lógicamente dentro de la lista de archivos.

Al iguel que en `commit` y en `getVersion`, aquí se bloquean los archivos del servidor para asegurar que un único cliente pueda modificarlos a la vez.

4. Conclusiones

Todos los integrantes del grupo han usado sistemas de versionado (**SVN** y **GIT**) como herramienta para llevar a cabo los trabajos prácticos de distintas materias. Sin embargo, hasta hace un mes nunca habían pensado la lógica requerida detrás de las funcionalidades que se ofrecen. Este trabajo fue una oportunidad para comprender cómo se pueden manejar distintas versiones de archivos dentro de un *filesystem*, y su interacción con unas cuantas *system calls* propias de **Minix**. Se realizó un trabajo de diseño elaborado para determinar la forma más óptima de almacenar los índices de los archivos y sus respectivas versiones. Se buscó en el proceso lograr una navegabilidad y persistencia de datos sencilla, y la habilidad de que varios usuarios interactúen con el sistema a la vez. Como producto final, se obtuvo una interpretación propia de un sistema de versionado concurrente que, a pesar de tener sus diferencias con sistemas populares como por ejemplo **GIT**, replica unas cuantas funcionalidades a su manera.