Mounting of Version Control Repositories RepoFS

Prashanth Mohan and Aanjhan R
{*prashmohan, aanjhan*}*@gmail.com*

***Abstract*— Version Control Repositories are run aplenty on the Internet. The clients for these VCS are often plagued with cryptic commands and access methods. Many of them also lack simple and intuitive user interfaces.**

This project aims to provide an easy to use mechanism of mounting any remote version control repository to a local directory. Thereby the usual UNIX commands and access mechanism can be used on the remote files. All File System calls are appropriately converted into the corresponding versioning functions.

***Index Terms*— File System, Version Control System**

I. Introduction

MOUNTING of a Version Control System holds many advantages to the common user. If one was to work on such a File System, all his work is automatically versioned. It is also of importance to developers who will be able to easily access repository files. Access to the repository files is made extremely easy. Some of the advantages of such a mounted Version Control Repository are:

- No need to specify the Version Control System type since it is determined dynamically based on the URL being mounted
- Ubiquitous access. All applications will be able to access the files directly from the repository, instead of only the VCS clients
- Implicit Versioning. Each modification of the file will be recorded in the version control repository
- Pluggable architecture. More modules supporting other VCS can be added at a later point of time
- Files copied and removed from the mount point is automatically translated in the version repository.
- Effective alternative to a Versioning File System

II. Overview

A. Version Control System

In today's world of commercial projects, Version or Revision Control [1] of source code is considered an essential Quality Management task. Especially in the Open Source world where contributions are made by numerous volunteers spread across the world, versioning of source code is indispensable. Version Control software provide various features like branches, tagging, locking, merging, conflict resolution, etc. Some advanced File Systems like ZFS, Wayback, ext3cow, etc provide for file versioning inherently, however, the topic of our discussion would be centralized version control repositories where, the version information itself is stored on a remote server.

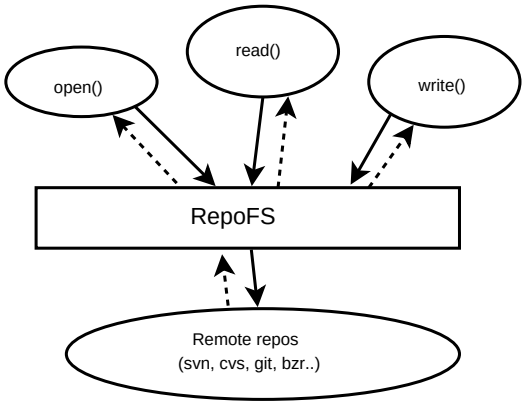

Fig. 1. Design of RepoFS

There are many Version Control Systems [1] like CVS, Subversion, Bazaar, Arch, Git, etc. Each one has evolved to address the limitations of the other.

B. File System Based VCS

Mounting of Version Control Systems (henceforth called VCS) provide numerous advantages to the lay user. There exist some solutions for mounting CVS repositories on a local directory [2]–[4]. Most of these solutions however provide only read-only access to the files. The advantages of a file system mounted versioning repository is the transparent mechanism for accessing its files. Another advantage of such a system is that the whole set of files need not be downloaded, but only those files which are read need to be downloaded from the repository [3]. This would have excellent implications in conditions where bandwidth is at a premium.

III. Design

The basic idea behind any User Level File System is that, the system calls that we access from our programs are nothing but glibc calls . By re-defining the behavior of these glibc calls, we can mount a File System from user space. Note in figure 1 that the kernel does not play any role. Some of the implementations which use this idea are Translators and FUSE (Refer §IV-A)

[1] http://en.wikipedia.org/wiki/List_of_revision_control_software

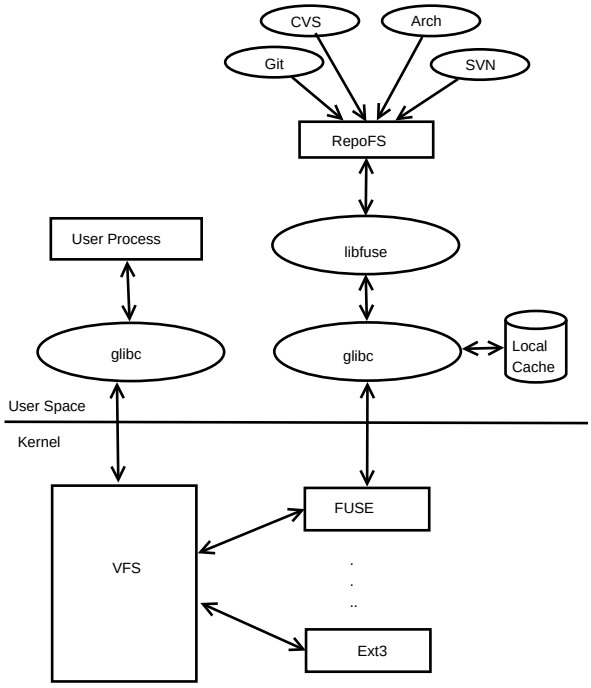

Fig. 2. Architecture of RepoFS using FUSE library

A. Plugin based Version System

The proposed RepoFS is designed to be generic and capable of mounting any kind of VCS. The logic and code for the respective VCSs will be defined in a plugin. The project itself will provide a framework for choosing the right plugin from the available choices and then register the appropriate call backs. The duty of RepoFS is to:

1) Identify the system which defines the repository being mounted
2) Check if the appropriate plugin is available (Offline or Online)
3) If it is available, then register the appropriate functions from the plugin for the file system access calls (Refer §IV-B). Read basic repository information, like the listing of the parent directory, so that the size of the mounted directory can be determined.
4) If the plugin is not available, attempt to download from the network and install (since everything is in user space). If the plugin is still not available report that the File System type is not supported and exit

B. Virtual Directories

Each time a repository is checked out, one would notice that there is an accompanying directory or file (one or more) along with the checked out source code, which is not actually part of the files under version control. This directory holds the revision information of the individual files and directories of the working copy. When we try to look at an online repository as an extension of a file system, this directory no longer holds any relevance.

We will reuse this directory to hold the meta data of the working copy. This will provide the backbone for other VCS commands like '`diff`', '`status`', etc. This directory will not exist physically, i.e. it will be a Virtual Directory [1], [5]. If suppose the user is creating a tarball of the mounted repository, then it is illogical that this virtual directory be part of the tarball. Hence, this directory will not be shown during a directory listing. However, the directory can be accessed by the chdir call. A similar virtual directory will exist for each subsequent subdirectory holding its own revision information.

There are various on on-disk File Systems like ext2, ext3, reiserfs, JFS, XFS, etc and there are memory based File Systems like procfs and devfs. The Virtual Directories will be a melange of the two implementation methods.

Let us take the case of an SVN (subversion repository) and work by using the '.svn' virtual directory as example.

1) versions: The different versions of the files are listed inside the directory called 'versions' and a sub-directory of the same name as the file itself. i.e. '`<mnt-point>/.svn/versions/<file-name>`' will list the different versions of the file based on the version number.

2) status: The 'status' of a working copy of the code will specify the details about which files have been modified, etc. This should be accessible by '`<mnt-point>/.svn/status`'.

3) log: The '`log`' command usually provides information about who made changes to the files, the accompanying messages and the revision numbers. Because there are 2 ways of invoking the '`log`' command; either on the directory or on the file.

- `<mnt-point>/.svn/.log` will be the alias of the '`svn log`' command which will provide the summary log of all the files.
- `<mnt-point>/.svn/log/<file-name>` will be the alias of the '`svn log <file-name>`' command which will show the log corresponding to that particular file.

4) diff: The '`diff`' command like the 'diff' utility shows how a file has changed over time. Similar to how the 'log' command was featured, the 'diff' command be accessed from the File System as follows:

- `<mnt-point>/.svn/.diff` will be the alias of running the '`svn diff`' on the directory.
- `<mnt-point>/.svn/diff/<file-name>` will be the alias of running the '`svn diff <file-name>`' command.

The diff itself is only a diff between the local version of the file and the HEAD version of the file. To get a diff between the different versions of the file, the `GNU diff` tool can be used to generate the diff between different versions of the file as mentioned in §III-B.1

C. Net File Sysntem

In order to access the network (which is essential to access the repository), we require that the File System driver be available in the user space. Hurd also provides a *libnetfs* which is used to provide some very commonly used network tasks.

The Hurd Extras page[2] and the FUSE home page list a number of file systems which use the Internet for its functioning. Some of them are SSHFS, FTPFS, etc. The Network File System (NFS) also uses similar concepts.

IV. Implementation Details

The file system is to be implemented in user-space, and there are many host Operating Systems on which this could be built on. We will look at the 'GNU/Hurd' and 'Linux' implementations, both of which are available in the Open Source domain.

The GNU/Hurd [7] Operating System is a micro kernel approach which uses the Mach[3] kernel. This OS is still largely a research OS and has not been adopted in the mainstream market. However, for our project we will choose to implement on top of the GNU/Hurd platform largely because of the power and flexibility that Translators (Refer §IV-A.2) provide us and the elegance of the micro kernel design. The Microkernel design also lets us stick to the architecture as defined in figure 1.

Linux on the other hand is an Operating System which is being increasingly adopted in both the server and the desktop markets. However, the monolithic nature of the kernel has caused some shortcomings. The FUSE (File Systems in User Space) project has been initiated to create an API for easy creating of User Space File Systems. The FUSE library is known to run on the Linux-2.4.x, Linux-2.6.x kernels and on FreeBSD. Work is also on to port the FUSE library to Hurd.

A. Libraries

1) FUSE: The FUSE[4] library has language bindings for many Object Oriented Languages including C++, Python, etc making development using FUSE modular and maintainable. FUSE works by accessing a kernel module through the VFS (See figure 2). However, the user level file systems will only access libfuse API.

2) Translators: Translators [6] are user processes attached to a local inode. The translator process can be thought to define the access methods for the local directory or file. Translators also include Network Translators, Symbolic Links, etc. However in our discussion here, we will restrict ourselves to Translators acting as File System drivers. The translator behaves likes a file or a directory depending on the command accessing it.

An active translator is a translator process attached to an inode and is currently in execution. Whereas, a passive translator is a translator which is attached to a local inode but is not in execution. Passive translators also have the additional advantage of being attached to an inode across multiple reboots of the machine. Hence, a configuration file or a mount on each reboot is not required. This is the required behavior of our File System since the user needn't be bothered about remounting the repository on each reboot.

[2]http://www.nongnu.org/hurdextras/
[3]http://en.wikipedia.org/wiki/Mach_microkernel
[4]http://fuse.sourceforge.net/

B. Access Mechanisms

The following file system mechanisms will be mapped to appropriate VCS mechanisms. We describe here a very abstract mechanism of how the function calls are supposed to behave. The behavior for independent systems will need to be defined separately.

1) creat(): Once a file is added to this mounted directory, now comes another decision point whether to reflect the addition on the repository or only update the local cache. For our project, we choose to reflect the additions on the main repository. We would typically to map the creat() call into a mix of 'add' and 'prop' commands. However, if the repository was to be mounted read-only (or in the case of the user not having commit access), then the creat() call would return an error for the lack of sufficient permissions. The directory holding the files would not have write permissions.

2) remove(): This is a more controversial call compared to the previous one since this could lead to potential data loss. However, since this is a versioning repository, the intensity of the loss would be slightly lesser. We choose to reflect the local changes on the main repository as in creat() call. Once again, if the user is mounting the repository in read-only mode or has insufficient permissions, this call will return an error.

3) readdir(): This system call should merely list the files in a given directory of the repository without actually downloading (checking out) the files to the local machine. A listing of the directory along with the file properties might or might not require a checkout of all the files. CVS requires the complete checkout of the directory and calculating the meta-data locally, whereas, SVN allows the `proplist` command which will list the file properties.

4) open(): This should increase the reference count of the file locally. However, the check out of the file is not done yet. This function will only check if the file exists in the repository. If it does not exist, it will return an error else, it will maintain a local reference count.

5) read(): This is the function call which will actually cause the downloading of the file to the local machine (unlike open()). The first call of read() will initiate the checkout of the file. Subsequent calls of read() will merely read it from the local cache. A time out value is associated with the read command. In case the network is down or the file cannot be retrieved, this function call should return an error.

6) readattr()/setattr(): These permission/attribute are created locally first. In case the upstream system allows for these attributes to be reflected, then it is committed during unmount of the File System. SVN for instance supports these calls using the `propdel, propedit, proplist, propset` commands.

7) write(): A repository may be mounted either read only or read write. This depends on the privileges of the user account. If an anonymous account or any other read only account is used to mount the File System, then the function calls to write will return an error.

However, if the repository is mounted by a user with full write permissions, then the write function will cause changes to be made to the local cache. The changes will not be made

upstream i.e. it will not be committed immediately. The new data of the file is maintained in cache.

Different Implementations of the CVS File System choose different approaches to the write problem. The most prevalent method has been to mount the repository as a Read Only file system [2], [3]. We will have to maintain a diff of the current file with the upstream copy.

8) close(): This is probably the most controversial of all the calls. We will have to make sure that there are no conflicts. In case of a conflict, the close call has 2 alternatives. It could either reflect the conflicts on the local cache of the file or it could simply return an error and require that the file system be remounted.

If the reference count is more than 1, it means that the cache of the same file is being accessed more than once at the same time. Now, if the file contents is modified out of the open process, it could cause a problem. Most intelligent editors like Emacs, will detect that the cache has changed and prompt the user to refresh the buffer. So, the former approach will be taken and the conflicts will be reflected on the local cache of the file. This might require that the upstream copy of the file be checked out again and the diff produced locally.

V. Existing Works

There have been implementations of a few of the above presented ideas. Stefan Siegl [3] has created an implementation of the CVS File System using GNU/Hurd Translators. There have also been some implementations of mounting CVS repositories as File Systems using the FUSE libraries [4], However almost all the implementations are plagued by one severe constraint – which is the handling of writes to the repository. Another shortcoming of the current solutions is that, they only offer support for the CVS system, and do not provide support for mounting a generic Version Control System.

VI. Conclusion

RepoFS provides a scalable and extendable architecture for mounting of Version Control Systems as a local File System. RepoFS will support not just one Version Control System but multiple Systems. It will be able to automatically update itself and download the required plugins for supporting new systems because it is entirely in User Space. RepoFS also provides for a well structured policy for managing mappings between File System calls and the corresponding functions in the Version Control Systems.

References

[1] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion*. O'Reilly Publications, 2004. [Online]. Available: www.svnbook.red-bean.com

[2] J. R. Nogueras, P. A. Portante, and W. Shi, "Presentation of a cvs repository as an sfs read-only file system," Master's thesis, Massachusetts Institute of Technology, December 2000.

[3] S. Siegl, *Virtual CVS filesystem translator for the GNU Hurd*. [Online]. Available: www.nongnu.org/hurdextras/

[4] P. Frank and L. Strojny, *CVS virtual file system*. [Online]. Available: www.sourceforge.net/projects/cvsfs/

[5] R. Pike, D. Presotto, and S. D. et al, "Plan 9 from bell labs," AT & T Bell Laboratories, Murray Hill, NJ, Tech. Rep., 1995.

[6] Debian. (2006) Translators. [Online]. Available: www.debian.org/ports/hurd/hurd-doc-translator

[7] M. I. Bushnell. (1994) The hurd: Towards a new strategy of os design. [Online]. Available: www.gnu.org/software/hurd/hurd.html