Presentation of a CVS Repository as an SFS Read-Only File System

Jorge Rafael Nogueras, Peter A. Portante, Wei Shi
{rafaeln, portante, shi}@mit.edu

Laboratory of Computer Science
Massachusetts Institute of Technology
545 Technology Square, Cambridge, MA 02139

December 7th, 2000

Abstract

We present a read-only file-system view of a Concurrent Versions System (CVS) repository, providing a useful mapping of familiar file-system semantics to CVS browsing operations. This mapping gives users a more natural way of browsing a CVS repository. Through the use of virtual directories, we transparently provide several commonly used CVS features, like revisions, history and tags, through file-system semantics. We use the Secure File System (SFS) framework for our NFS3 file-system implementation, gaining the benefits of the SFS security model.

1 Introduction

The Concurrent Versions System (CVS) is a version control system: it works as a database where different versions of source files can be stored and later retrieved. It is widely used for keeping repositories of code that can be accessed by different developers simultaneously. It can be used to store text or binary file types. However, it is optimized for handling text files.

In this paper we describe a read-only file system interface to CVS that allows users to access a repository using natural file-system semantics.

1.1 Why is CVS popular?

The reason CVS is so popular is because it facilitates project management by allowing developers to “check out” (retrieve) code stored in the repository, perform local changes to it, and later commit these changes to the repository for others to see. It reduces the necessary coordination between developers: files can be "checked out" by more than one user at a time, without resorting to locking the files while changes are being made.

Perhaps one of the reasons why CVS has flourished is because it supports, and even encourages, the open, evolutionary development methods favored by free software [1], which greatly benefits from decentralized management. This is why many free projects, especially those with large, distributed development teams, store their source code in a CVS repository (and the source code for CVS itself is no exception).

It must be noted, however, that although CVS aids in project management, it is not a substitute for it [2]. Schedules, merge points, branch names and release dates are considerations that must be coordinated in conjunction with all the developers. Even though CVS provides tags and branches to aid the tracking changes, the decisions associated with when to tag and when to branch are outside the scope of CVS and this file system.

1.2 How is it used?

The editing cycle using CVS beings when the user "checks out" her own copy of the files from the repository into her local directory; these checked-out files are referred to as her "working copy." Changes performed on the user's working copy do not affect the public repository. After the user has finished the changes to the working copy, she must commit, or "check in," her changes back into the CVS repository. In reality this is a two-step process: before committing a file, the user must make sure that no one has made any changes to the public repository that would be overwritten by the working copy. For this reason, the user should update her working copy with the latest

version in the repository, possibly merging the file's contents with that of the repository.

While CVS is usually adept when performing merges between different versions of a file, there are occasions in which the versions conflict in a way that requires user intervention. When that happens, CVS flags the file as having a conflict, which must be manually resolved before being finally checked into the repository.

1.3 What is CVS lacking?

Although CVS provides a way of maintaining a repository of files (much like a local disk), one cannot access these files in a way that would seem natural (users access CVS repositories using commands typed at a command prompt). There are graphical user interfaces for CVS (see Section 4), but it is more natural for developers to interact with their source files using command-line tools that work using file-system semantics. It would therefore be valuable to be able to "browse" through the repository by going to a particular directory, asking for its contents, changing to another directory, maybe printing the contents of a file, et cetera. Files should look and feel as though they actually reside on a file system, and file system semantics should be used to operate on them.

1.4 What are the problems that we solve?

Currently, developers using CVS must execute a "checkout" command to get a copy of a given module to browse its contents. For larger and more complex projects, there may be many released versions of the software. This would require the developer to execute multiple CVS "checkout" commands in order to retrieve the contents from various releases, even if the user only wants to look at one file's contents. It may not be convenient to populate a local file system when accessing such large projects.

Also, one of the standard methods of accessing a CVS repository from across the network is the password server, or "pserver." It works with a simple password file, which contains usernames and encrypted passwords for authorized CVS users. The disadvantage of using this authentication method is that passwords have to be transmitted as clear-text over the network: the "pserver" does not have the option to encrypt the traffic (passwords are trivially encrypted, but this would not thwart an active attacker, only a casual observer). This may compromise the security of the repository if an attacker is examining packets on the network and thus gleans user and password information. The attacker could then log in using the "pserver" method and modify the files in the repository: what is even worse, after having succeeded in gaining repository access, the attacker may effectively gain access to the whole system, since CVS has features that execute arbitrary commands on the system.

Aside from the "pserver" method, there are options that allow for Kerberos authentication (both versions 4 and 5). Using Kerberos, it is possible to authenticate a user, set up a secure connection, and transmit the data encrypted. However, the Kerberos framework requires a centralized server that is in charge of all authentication requests. Furthermore, it requires all possible users to have previously registered with the centralized server.

1.4.1 What is our solution?

We solve these problems by using the SFS framework to implement a Network File System 3 (NFS3) server that will provide a read-only file system view of a CVS repository. The entire CVS repository can be mounted as a remote file system and accessed across the network. The user is provided a general file-system view of the CVS repository and she can choose which modules and versions to view by moving around in the file system. Typical file-system operations can be applied to these read-only files, such as **`diff3`**, **`vi`**, **`cmp`**, **`cksum`**, et cetera. Per-file CVS history information is also accessible through the file system.

Although using the SFS file-system framework guided our implementation to some degree, we must note that our code was designed to interact directly with a NFS3 layer. That is, while it works well under the SFS framework (which also relies on NFS3), it is independent of it and can be used with any NFS3 framework. However, as we have

discussed before, SFS provides secure data transfer, which NFS3 lacks, so it makes sense to use SFS as the underlying networked file system framework.

1.4.2 What does SFS bring?

The Secure File System (SFS) [3] provides transparent and secure access to a remote file system. It prevents many of the vulnerabilities caused by today's insecure network file system protocols. It makes file sharing across administrative realms trivial, lettings users access files from anywhere and share files with anyone.

SFS decouples key management from file system security. SFS filenames are constructed by embedding public key information in the path itself (they are called "self-certifying pathnames"). SFS needs no separate key management machinery to communicate securely with file servers, nor does it need a centralized authentication server like Kerberos does.

If one desires to export an SFS file system to the public, the SFS server software is run, allowing remote users to mount the exported directory and access it as though it were a local file system. Security is provided by symmetric and asymmetric encryption schemes that ensure the data transferred cannot be deciphered by anyone looking at network traffic.

Using the SFS framework for exporting a file system, which provides excellent security, is a proper base from which to develop a CVS read-only file system.

In Section 2 we will explore the design of our project, talking specifically about the file-system interface that is presented to the user; in Section 3 we will give a few implementation details; in Section 4 we will talk about related work; in Section 5 we give ideas for future work on the subject; and finally in Section 6 we will offer some brief conclusions.

2 Design

We chose to implement a read-only file-system for CVS, since browsing a CVS database can be awkward and unnatural. We did not implement a read-write file-system since the file system semantic mapping to CVS commands required more time than we had been given (See section 5.1). Several principles guided our CVS read-only file system design:

- **Transparency:** the CVS file system should behave in a manner consistent with any read-only file system; the SFS client software should not change, and should not be even aware that it is working with a CVS repository
- **Ease-of-use:** the user should be able to navigate the repository using familiar file system commands
- **Minimal framework changes:** the user-mode server should not require extensive changes to work with the CVS
- **Extensibility:** the implementation should lend itself to future changes by having an abstract design that does not restrict the file-system view in any way

We do not address performance issues: we are only concerned with the conceptual design and the usability of the resulting interface. We outline the different aspects of our project design in the next sub-sections.

2.1 Virtual Directories

In order to meet our goals for this file system, we use a concept called *virtual directories* to provide the transparency and ease-of-use required. A virtual directory is a directory that is created or interpreted by the file system itself and has no real counterpart in the repository. They exist solely to expose some aspect of CVS functionality in the file-system, relieving the user from having to memorize and type commands, instead using file-system semantics. The idea of a virtual directory is akin to the way special files are used in the Plan 9 operating system [4]. It often uses special-purpose files to endow the file system with a particular function (like networking and accessing other system resources).

The use of virtual directories is quite pervasive throughout our framework. The obvious drawback of virtual directories is that whatever name is chosen might already be in use by the repository. This is similar to the problem of a

working directory containing a bona-fide directory named **CVS** (CVS creates a directory called **CVS** in every subdirectory of a user's working copy for its own purposes). If there is a real directory with a name that clashes with a virtual directory, the latter will simply be inaccessible until the real directory is either renamed or removed.

2.2 Namespace Mapping

One of the first things we had to decide was how to map the contents of the CVS repository to a file system. For the most part, the internal structure of a CVS repository is stored using directories and files: our file system simply exposes that existing structure. This provides for ease-of-use since a user of an existing CVS repository will already be familiar with the structure.

CVS has three particular features, repository-wide tags and branches, file-specific revisions and revision history, which are accessible transparently in our file system design.

2.2.1 Tag view

Tags are markers in CVS that bind files (with their respective versions) together with a common name. They can be thought of as symbolic names that represent the state of a project at a specific moment in time; they make it easier, for example, to revert to or view the last stable revision of a project.

We expose tags as virtual directories in the root directory of our file system. All tags in the CVS repository are visible as directories from the root of the file system. This way, changing to a particular directory from the root would show the directory tree of the repository as it looks for that tagged version. For instance:

```
$ cd /cvs/
$ ls -F

HEAD/        1_3_00_rev/    bugfix
tag1/
```

(For the purpose of simplifying our discussion, the **/cvs/** directory is actually a symbolic link to the full self-certifying SFS pathname representing the exported CVS repository: the remaining examples assume this link is in place.) We can see that **1_3_00_rev**, **bugfix**, and **tag1** are names of tags that appear throughout the repository. They are shown as directories so the user can see the revision they represent by simply changing directory.

Note there is a directory called **HEAD** shown: this is a virtual directory that will always appear at the top-most directory level, whether or not other tags are present. It represents the repository as it looks in its latest incarnation (which may or may not have been tagged with a specific name). The name comes from a special tag that CVS uses in some contexts to refer to the end of the main development branch, or the "main trunk."

So, when the user then types:

```
$ cd HEAD
$ ls -F
```

the list of all the subdirectories of the CVS repository will be shown:

```
xemacs/      kde/    class-project/
```

and changing directory appropriately to another tag exposes the corresponding directory structure as it appears in the repository for that tag.

The file system exposes all repository tags at the top-most level of the directory structure to provide for easy access to the tagged information. As source code projects evolve and become more complex, users of CVS repositories are relying on tagged branches to help manage the state of a project more effectively. This file-system feature targets large CVS repositories with complex branching. The main trunk of a CVS repository is used mainly as a synchronization point between branches, and often not for active development. Although access to the main trunk is still useful (and provided through the **HEAD** virtual directory), it may actually be overshadowed by more frequent use of branches and tags.

2.2.2 Revisions and Branches

Another useful feature to expose to the file-system view is the files' version numbers, or revisions, as well as branches and branch revision hierarchy. Each time a file is committed, or checked into the repository after being edited, its revision number

is increased by one. This means that there is always a way of seeing how an earlier revision of a file looked.

When a branch is created, there is a special type of revision number associated with that branch and the revisions therein will evolve separately from the revisions on the main branch of the repository (this may be useful, for example, to do development in parallel, perhaps to fix a bug in a previously released version concurrent with main-branch development). It is noteworthy that branches may be created off of existing branches, thus creating an arbitrarily complex branch hierarchy.

We incorporate these concepts into a file-system view as follows: inside each directory in the repository, a virtual directory called **`.fi`** (short for "file information") will exist; its contents are the same as the current directory, but all files appear as directories (explained later). This virtual **`.fi`** directory will not show up in directory listings, however; this facilitates doing operations that involve the whole directory structure; for instance, **`tar`**'ing a directory and its subdirectories. If the **`.fi`** directory were returned in directory listings, the "tar" program would include the files inside it, adding them to the archive when the user intended to include only those files actually in her working copy. Even though the **`.fi`** directory is not shown in directory listings, users can change directory to it and list its contents.

The **`.fi`** virtual directory is structured as follows: all files which appear in its parent directory, appear as directories in the virtual directory; each of those directories in turn contain the file's different revisions as files named by their revision numbers, with tag names as symbolic links, as well as directories representing the branch hierarchy. An example will help in illustrating this scheme:

```
$ cd /cvs/HEAD/xemacs
$ ls -F

file1.c      README

$ cd .fi
$ ls -F

file1.c/     README/

$ cd file1.c
$ ls -F

.log         1.1            1.2
1.3          1.2.2/         rev-1@
bugfix@
```

(The use of the **`.log`** file will be explained in the next section). By examining these files, the user can see how each file looked at a specific revision number. As a concrete example, versions **`1.2`** and **`1.3`** of file **`file1.c`** can be **`diff`**'ed (compared) like so:

```
$ cd /cvs/HEAD/xemacs
$ diff .fi/file1.c/1.2 .fi/file1.c/1.3
```

We represent tags in this scheme as symbolic links. The **`rev-1`** is a symbolic link pointing to the appropriate revision number (so for example, printing out **`rev-1`** would be the same as printing out the revision tagged with that name).

Note that there is also a directory called **`1.2.2`**: this 3-digit revision number is a branch identifier. CVS uses revisions numbers with an odd number of digits to represent branch revisions. They also have a tag name that corresponds to that revision number; in this case it is shown as the symbolic link **`bugfix`**.

Thus, all branches in which the file exists will have their own subdirectory:

```
$ cd 1.2.2
$ ls -F

1.2.2.1      1.2.2.2        1.2.2.3
```

would show the revision numbers for the file on that branch. The file **`1.2.2.3`**, for example, would contain the latest revision of **`file1.c`** on branch **`1.2.2`**.

2.2.3 History

Another useful feature of CVS is its ability to store log messages with each file as changes are committed to the repository. Whenever a user commits her changes to the repository, she must type a log message that should explain the nature of the changes that were made to the files. These log messages should aid other users in figuring out the changes that a file has gone through and the reason for said changes. The list of all the log messages attached to a file through its different revisions is called its history.

We believe it is useful to expose the history of these log messages in the file-system view. That is why in the virtual **.fi** directory there is a **.log** file that lists the file's log messages. For instance:

```
$ cd /cvs/HEAD/xemacs/.fi
$ ls -F

file1.c/     README/

$ cd file1.c
$ ls -F

.log         1.1            1.2
1.3          1.2.2/         rev-1@
bugfix@
```

and printing out the **.log** file would display all the log messages for **file1.c**.

2.3 File-system Semantics

Having described how CVS repository elements are mapped to a file-system view, there are a few more design issues to consider.

2.3.1 File status

For each directory and file in the file system, we must generate creation, access and modification times that make sense. For a given directory, we use the actual file times of the directory in the CVS repository itself. For a given revision of a file, the creation, access and modification times are taken from the CVS revision creation time.

2.3.2 Updates to a CVS Repository

A CVS repository is dynamic: directories and files can be added and removed; new revisions are continuously being generated. We provide for the dynamic nature of CVS repositories by automatically updating the read-only file system with the changed information. This behavior is no different from making updates to a local Unix file system visible to remote clients.

2.3.3 NFS3 File Handles

One feature of a CVS repository is that a given file is never actually removed. Rather, a record of the removal is kept so that a particular revision of the file in the past can always be accessed. This means that a file handle for a given revision will always be valid. Checking in a new version of a file will create another revision, leaving previous revisions intact. So if a user is viewing a tagged hierarchy, which is not a branch tag, a new revision checked in for a given file will not disturb the tagged hierarchy

For a branch tag view, the file system presents the latest revision for each file in that branch. As updates are made to a branch, the file system will automatically incorporate them. As a result, an existing file handle will become stale when a new revision is made to the file on that branch and subsequent accesses will see the most recent revision. Note that the main line tagged with **HEAD** also behaves this way even though it is not considered a branch.

2.4 Authentication

CVS requires the user who is performing an operation to authenticate herself somehow: in the case of a user executing a CVS command locally, it verifies that she has the appropriate access to the repository. In our system, the user that starts the SFS server running performs the CVS operations on behalf of the client user. Thus, any client will be able to access those parts of the repository available to the server user (and then, of course, performing read-only operations).

This means that if there is a reason why the read-only access to the CVS repository should be restricted to only some authorized users (instead of everyone who knows the server's self-certifying pathname), some user authentication must be done at the SFS level. It is, however, outside the scope of our work.

3 Implementation

We implemented our CVS read-only file system using the SFS user-mode server as our framework. This provided us with a solid NFS3 implementation and excellent security infrastructure. We augmented the user-mode server so that it could serve either a normal Unix file system or our CVS read-only file system, selected via a simple configuration option. We

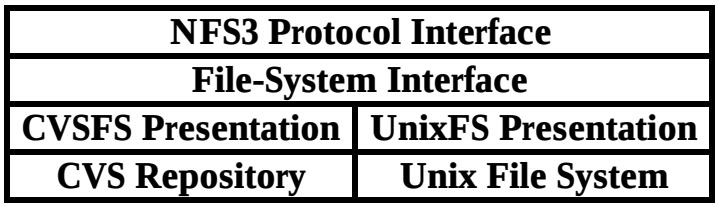

Figure 1: Layering of the SFS user-mode server

chose to modify the server in such a way that any other file-system could be added in a similar fashion. We made minimal framework changes to the existing server code, allowing us to leverage its NFS3 protocol implementation for our new file system.

3.1 Interface Layering

We define a series of interface layers in order to implement our file system with a minimal amount of change to the existing framework. As shown in Figure 1, we logically separated the user-mode server into four layers, each with distinct responsibilities and well-defined interfaces. The top two layers, which we refer to as the *abstract* layers, are shared by all file system implementations within the server, while the bottom two layers, referred to as *concrete* layers, are provided by the individual file system implementations. This construction allows us to leverage the existing NFS3 protocol implementation for all file systems.

3.1.1 NFS3 Protocol Interface Layer

The SFS user-mode server conveniently provides the definition of our first layer, the NFS3 protocol interface. We chose to keep this layer intact, not making any semantic changes to it, so that all file system implementations will behave consistently and so that lower layers need not worry about NFS3 semantics. This keeps our framework changes to a minimum and allows us to focus on the concrete file system layers.

3.1.2 File-System Interface Layer

The NFS3 protocol layer was designed to work with traditional Unix file-system interfaces. We define an abstract file-system interface layer, with identical semantics, as the boundary between the NFS3 protocol layer and a particular file-system's concrete layer. No changes where made to the NFS3 protocol layer as a result of adding support for our new file system, other than the minimal changes needed to utilize this abstract layer. To the NFS3 protocol layer, all file-systems have the same semantics. This prevents the protocol layer from having to know the details of a particular file-system implementation. In turn, the concrete layers for each file-system do not have to know NFS3 semantics.

Additionally, this layer provides the benefit of being able to server more than one file-system at a time, since it can dispatch to the particular file system's concrete layer as needed. This allows an instance of an SFS user-mode server to serve both a traditional Unix file system and our CVS read-only file system at the same time.

3.1.3 Concrete File-System Layers

The concrete file-system layers implement the abstract file-system interface, providing the functionality for the particular file-system they represent. There are two distinct layers within the concrete layers: the top one we refer to as the *presentation* layer and the bottom one as the *raw* layer.

Our modifications to the SFS user-mode server provide two concrete layer implementations. The first is the pre-existing Unix file system implementation, which we will refer to as *UnixFS*; the second is our new CVS read-only file system implementation, referred to as *CVSFS*.

The UnixFS concrete layers are quite simple. The operating system's file-system interface *is* the raw layer, with the presentation layer being a simple translation of our abstract file-system interface

semantics to and from the operating system semantics. These layers provide the behavior of the pre-existing SFS server.

Our CVS read-only file system layers are more intricate since they need to fabricate the actual file system for the repository to be served. The following sections describe the details of each layer. We present the raw layer first, followed by the presentation layer, to make it easier to understand the presentation layer implementation.

3.1.3.1 CVS Raw Layer

The CVS raw file-system layer provides the presentation layer access to a CVS repository. There are two fundamental pieces of information provided by this layer: the file-system hierarchy of the repository and the list of all tags in the repository.

To provide the file-system hierarchy of the repository, the raw layer maps it into a pseudo file system hierarchy we call the *object* hierarchy. Each directory and file in the repository has a corresponding object in the object hierarchy. A given object in the hierarchy provides the necessary methods for accessing that object's data in the repository.

Each file object in the hierarchy provides three pieces of information from the repository: the list of revisions and tags associated with the file, the CVS history log entries and the contents of the file for a given revision. Each directory object in the hierarchy simply provides the list of all files and directories contained within it.

3.1.3.2 CVS Presentation Layer

The CVS presentation layer implements the file-system semantics outlined in Section 2. In particular, all virtual directories, which do not have a real counterpart in the repository, are fabricated in this layer. The basic file-system hierarchy is built from the object hierarchy provided by the raw layer. The repository-wide tag list provided by the raw layer is turned into the virtual directories located at the root of the CVS file-system hierarchy. Underneath each of these virtual directories, the object hierarchy is then replicated and displayed. It must also display the file information virtual directories and provide their semantics.

For instance, as described in Section 2, suppose the user types the following:

```
$ cat /cvs/HEAD/xemacs/.fi/file1.c/1.1
```

This means that the user wants to see version **`1.1`** of **`file1.c`** in the directory **`xemacs`**, from the main branch of the revision tree. The presentation layer first validates that **`HEAD`** is a valid tag, checks that **`xemacs/file1.c`** exists in the object hierarchy, and then requests the contents for revision **`1.1`** of **`file1.c`**.

3.2 Implementation Specifics

We chose to use the SFS user-mode server for our file system implementation because a kernel-level server is not a good fit for a CVS database. The existing CVS code can be used from user-mode only. The level of difficulty to make it work in some form from within the kernel is beyond the scope of our work. By using the user-mode server, we also gain the benefit of reusing much the existing code.

We augmented the user-mode server's configuration file syntax so that a particular file system can be specified. The existing syntax we left as the default, and added an argument to the file system export option to request a specific concrete file-system.

We based our file-system on CVS V1.11 [5] since it was the latest revision available at the time. The source code for CVS is organized as one large executable containing all of the necessary functionality. We chose to rework the code somewhat to create an API for our needs. This was relatively easy to do, and helped us to keep the behavior of our file-system close to that of CVS' behavior.

The requirements of a read-only file-system place a few simple demands on the API we needed to develop. We needed to be able to discover the directory hierarchy and the files contained therein, fetch the list of tags and revisions for each file, get the contents of a file given a tag or revision, and retrieve the log history for each file.

4 Related Work

Not surprisingly, CVS' ubiquity has prompted many developers to create enhancements and add-ons to CVS. As far as we know, however, ours is the first attempt to expose CVS as a file system, even if it is only a read-only view of a repository.

We have encountered, however, several other projects that aim to improve a CVS repository's accessibility. For instance, there is CVSWeb [6], originally created as a Perl script by Bill Fenner and currently maintained by Henner Zeller: as its name suggests, it makes a CVS repository available through the web using any browser. In many ways it shares a common goal with our read-only file system, since it provides a read-only way of examining a repository. While it is quite effective, allowing common operations such as log browsing and viewing differences between versions, by definition, a user is limited to those operations presented in the web interface. A file system interface, on the other hand, allows any arbitrary operation to be performed on the files (like printing, **`grep`**'ing, copying, line counting, et cetera).

The SCVS, or "Secure CVS", project [7] attempts to address some of the shortcomings in CVS' security model. It basically starts an SSH (Secure shell) tunnel and routes all the CVS traffic through it. In general, when giving users remote access to a CVS repository using the "pserver" method, passwords are transmitted practically in the clear over the network. While SCVS addresses this security weakness in CVS, it does nothing to ameliorate the interface to the system.

There are many other projects that aim to provide a graphical user interface (GUI) to CVS: gCVS [8], jCVS [9], tkCVS [10] and WinCVS [11], among several others. While certainly improving CVS' interface by making it easier and more intuitive than a command-line interface, these enhancements do not bestow CVS with the flexibility of file-system semantics.

A commercial version control system, Rational's ClearCase [12], also provides a file system view of a source code database. The user can create a view and edit the configuration specification to specify which branch and label (similar to tags in CVS) to view. The user does not have to check out the entire module to view the source tree: however, ClearCase does not provide a file system view of the entire source code database as we do. If a user wants to view the source tree of a different branch or label, she would have to create different views or switch the specification of the current view [13]. Although it does have a GUI tool to view the relationship of branches and labels, it does not provide a file system view of the history of the code base. On the other hand, our **`.fi`** directory presents a natural file system view of the evolution of revisions and branches.

5 Future Work

There are several enhancements to our project that would improve its overall usability.

5.1 Read-write Repository

A natural progression would be to move from a read-only file system to a read-write file system, allowing the user to commit changes back to the repository, not only view its contents. We gave a lot of thought to this possibility, though compelling, we decided it was fraught with complications that could not be successfully dealt with in the timeframe we were given. Several issues made it particularly hard:

- How and when should edited versions of a file be checked into the repository? Clearly, files cannot be automatically checked in every time they are saved or every time an editing session ends. By definition, the check-in process is totally asynchronous and has to be left completely at the discretion of the user. Thus, the file system would have to provide a way for the user to check in a file, perhaps by having a virtual **`.COMMIT`** directory where files are copied when the user wants to check them in.
- How are files updated? There has to be a method (which again must be asynchronous and at the user's discretion) by which changes made to the repository are merged into the local working copy. Since this may cause conflicts that must be manually resolved, it is clear that the system cannot update files arbitrarily: the

user must decide when an update is appropriate. Like with committing a file, one can envision a virtual **.UPDATE** directory, and to specify that a file should be updated it is simply copied into that directory.

These considerations would have made the design more complex: first of all, there would have been a need for an intelligent, CVS-aware agent in the client. In our current implementation, the client is blissfully unaware of any CVS semantics and as such, needs not be changed in any way: it is simply given file and directory information by the server and it displays it to the user. Thus, the client cannot, and should not be able to, tell the difference between real directories contained in the repository and virtual ones. This fact greatly simplified the design and implementation of our system.

5.2 Project View

In our discussions we realized that there are several ways of exposing tags through a file-system interface, each with its own strengths and weaknesses. While we chose the "tag view" for our implementation, it is worthwhile to discuss an alternative view we had considered and how it might fit in a future version.

The *project view* of a repository is characterized by root directory containing the directories of the repository and not all of the repository tags. For instance, we would observe the following:

```
$ cd /cvs/
$ ls -F

.TAGS/       xemacs/        kde/
class-project/
```

and changing to each directory reveals the structure contained therein (it implicitly shows the latest revision of each file in the repository). However, since we also wish to expose tags in this file-system view, we would create the virtual directory called **.TAGS** containing the names of all the tags in the repository as directories, much like in the tag view. In other words, changing directory to **.TAGS** yields:

```
$ cd .TAGS
$ ls -F

1-3-00-rev/  bugfix/        tag1/
```

and changing to any one of those directories reveals the rest of the repository's hierarchy at the revision number corresponding to the tag.

We feel that a possible addition to our project would be the incorporation of the "project view" coexisting with the "tag view" that was actually implemented. One can visualize having the following two virtual directories at the top-most level of the repository structure:

```
$ cd /cvs/
$ ls -F

tag-view/    project-view/
```

where the contents of each directory would follow the format of each type of view. This way the user could choose whichever view she is more comfortable with, or which one makes more sense with the way that the repository is being viewed. This also allows for other views to be incorporated in the same file structure.

5.3 Extra CVS Features

CVS contains more features than we have exposed in our project. As an example, CVS has a feature called "watches," which aid in the synchronization between developers. A user may register an interest in a file using watches, and when another developer declares she is editing a file, the first user is given an indication of this fact.

Commands such as these may be implemented by having virtual directories with appropriate names (which should not be displayed in directory listings). To apply a command to a particular file, it should be copied to the directory; for instance:

```
$ cd /cvs/HEAD/xemacs
$ ls -F

file1.c      README

$ cp file1.c .WATCH/
```

to access this command. A possible drawback of using different virtual directories to expose these features is that it increases the likelihood of them clashing with bona-fide directories in the repository. However, this chance may be quite

small and the advantage given to the system would outweigh this minor annoyance.

6 Conclusions

We have presented a read-only file-system view of a CVS repository, providing a useful mapping of familiar file-system semantics to CVS browsing operations. This mapping gives users a more natural way of browsing a CVS repository. Through the use of *virtual directories*, we transparently provided several commonly used CVS features, like revisions, history and tags, through file-system semantics. We used the SFS framework for our NFS3 file-system implementation, gaining the benefits of the SFS security model. Also, our abstract layer design enables both CVS repositories and file-systems exports by the same executable image.

References

[1] K. Fogel, *Open Source Development with CVS*. The Coriolis Group, LLC: Scottsdale, Arizona, USA (1999).

[2] P. Cederqvist et al, "Version Management with CVS," http://www.cvshome.org/docs/cvspdf/cvs1_11.pdf (2000)

[3] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. "Separating key management from file system security," *Proc. SOSP*, pages 124-139, December 1999.

[4] Rob Pike, Dave Presotto, Sear Dorward et al. "Plan 9 from Bell Labs," AT&T Bell Laboratories, Murray Hill, NJ. (1995).

[5] CVS main site, http://www.cvshome.org

[6] CVSWeb, http://stud.fh-heilbronn.de/~zeller/cgi/cvsweb.cgi/ (2000).

[7] SCVS, http://cuba.xs4all.nl/~tim/scvs/ (2000).

[8] gCVS, http://www.arachne.org/software/gcvs/ (2000).

[9] jCVS, http://www.jcvs.org/ (2000).

[10] tkCVS, http://www.neosoft.com/tcl/ftparchive/sorted/apps/tkcvs-6.0/ (2000).

[11] WinCVS, http://www.wincvs.org/ (2000).

[12] ClearCase, http://www.rational.com/products/clearcase/index.jsp (2000).

[13] *ClearCase Reference Manual*, UNIX Edition Release 2.0, ATRIA Inc.