CVS RFPK HOWTO

CVS a widely used tool for managing revisions to documents is now available to RFPK Team Members. CVS is especially useful when a number of authors or developers work on the same set of documents, especially when these collaborators are not always located in the same office or the same campus. This tutorial provides background on revision control systems in general, and shows how to get started with CVS, in particular.

The original version of CVS presents a command-line interface to the user. Much of this document is a description of that interface. A GUI interface, called *cervisia*, is now also available, and is covered in the last section of this tutorial.

Table of Contents

Introduction **1**
CVS and other Revision Control Systems **1**
Repositories **2**
Environment Variables for Repository Access **3**
The Command Line Interface **4**
The Cervisia GUI **9**
Some Server Administration **10**

Introduction

Brad Bell and Ranjan Dash have started a new project that will include text documents as well as computer source code. To manage revisions to their documents, they have decided to use a system known as CVS rather than Microsoft Visual SourceSafe, which is currently being used by the Software Team to manage revisions to SPK.

CVS is a free, open source product used to manage revisions for many important software projects, including Apache, Mozilla, and Linux. It is especially useful when multiple authors, geographically dispersed, collaborate on the same documents. As configured at RFPK, CVS can be accessed with excellent security from anywhere that there is an Internet connection.

CVS is available to anyone in RFPK who has a need to manage revisions to their text documents. This paper contains information specific to the RFPK installation of CVS. Complete information is available in two manuals, copies of which are stored on the RFPK internal website. These are *Version Management with CVS* [1], by Per Cederqvist and *The CVS Book*, [2] *Open Source Development with CVS,* by Karl Fogel. If you have any questions, please feel free to contact Alan Westhagen (afw@u.washington.edu).

The CVS repository is on a Linux system, which has the Internet domain name whitechuck.rfpk.washington.edu . In order to use the CVS repositories on this system, you need to be installed as one of its users. Please contact Mike Macaulay (macaulay@apl.washington.edu) to add your login to whitechuck .

CVS and other Revision Control Systems

CVS is an example of systems variously referred to as *revision control* systems, *source control* systems, or *version control* systems. Widely used revision control systems include SCCS, RCS, Microsoft Visual SourceSafe (VSS) and CVS. All of these systems were designed to work very much as does SCCS, the senior member of the group,

which was developed at Bell Labs in the 1970s. Like SCCS, they all store revisions as differences between the revised version and the original, rather than as complete copies of each version, using the algorithms originally incorporated in SCCS.

In one important respect, CVS differs from the others. With all of the other systems, only one user is allowed to work on a particular document at any one time. When a user wants to make changes to a text, she must first *check it out*, which means that the revision control system gives her the exclusive right to make changes until she subsequently *checks in* her modifications and thus relinquishes exclusive control. If someone else has checked out the document that she wants to change, she must wait until that person relinquishes it. If she is slow to make her changes, others must wait for her.

CVS, too, was originally designed to incorporate the exclusive checkout model. This strategy was abandoned, however. The change came about in response to a new set of requirements that arose primarily as a result of the rapid expansion of the Internet. Previously, software development groups had almost always consisted of individuals working in the same office or the same building. In fact, the leading software development methodologies required this. When you are in the same office, it is relatively easy to negotiate directly with your co-workers about who should have exclusive modification access to a particular document. When, conversely, you have people working on different continents and in different time-zones and who communicate asynchronously via email, it may take days to get a co-worker to relinquish control. Similarly, with frequent travel, it became a necessity for team members to continue to work at hotels with unpredictable Internet connectivity, or even on airplanes or in airports. The exclusive checkout model had become an impediment to productivity in the new global work place.

The break-through came with the reversal of the checkout model. When a user checks out a project from CVS, she gets a complete working copy on her own machine. She is free to make changes to any of the files in her working copy. If, for example, her working copy is on a laptop, she can take her laptop off-line, travel to a conference or to her vacation home and continue to work on her project, secure in the knowledge that just because her computer is temporarily cut off from the Internet or just because she can only devote a small part of the day to the project, she is in no way impeding her collaborators. When her computer comes back on line, whether by dial-up connection, connection at an Internet cafe, or the connection at her office, she can quickly have her working copy updated with any changes her co-workers have made while she was cut off from the network.

What happens if the user's changes conflict with changes that have been committed by her collaborators in the mean time? This, of course, is the situation that the exclusive checkout model avoids. With the CVS model, however, it really is not a problem. When the user executes the CVS **commit** command to incorporate her changes in the repository, CVS will refuse, informing her that there are potential conflicts and that she will need to perform an **update**. The update merges the current copy in the CVS repository with her working copy, highlighting the differences. She can then edit her working copy to resolve any differences. When she once again runs the CVS **commit** command, her revised conflict-free revision will then be accepted, unless, of course, one of her colleagues has committed additional changes while she was resolving the previous ones, in which case the resolution process may require another iteration.

Occasionally having to enter into a conflict resolution process is the price users of CVS pay for the freedom to make revisions whenever and wherever they can be made most effectively. Especially given the importance of collaboration with partners at other institutions within the academic and scientific world, CVS will be a valuable addition to RFPK's set of resources.

Repositories

All of the documents under revision control are stored in a *repository* on a server. CVS

permits several repositories on a single server. At present, we have defined two. One repository is for the work being done by Ranjan and Brad. The other is for certain documents of the Software Team, including this tutorial.

The repository set up for Ranjan and Brad has its root in the directory located at the absolute path address `/usr/local/cvs_repos/rfpk/r1` on a server with domain name whitechuck.rfpk.washington.edu The other repository on whitechuck has its root in the directory /usr/local/cvs_repos/rfpk/r2 .

Within a repository, sets of documents are organized as modules. A module could comprise all the source code for a computer application, or all the source code for a part of the application. It could be all the documents required for an academic paper, or all the chapters of a book.

Within the operating system of the computer, the repository is a directory hierarchy. A module name can be associated with any directory in the hierarchy, with a single file, or even with a list of files. For example, the first module added to the r1 repository was the module called *muscle*, so named because it holds a paper that Ranjan and Brad are collaborating on which is about oxygen uptake in muscles.

Environment Variables for Repository Access

At present, the only way to use CVS at RFPK is from a Unix of Linux computer, or from the Cygwin environment on a Windows computer. In either case, the initial setup is the same.

All access to CVS at RFPK is via *ssh*, the secure shell. When you issue a CVS command from the command line on your workstation, the local copy of the **cvs** program communicates with the server via an encrypted channel that it calls upon **ssh** to provide. The command is actually executed on the server, with data being piped through the encrypted channel between the server and your workstation. This is, arguably, the most secure arrangement possible on the Internet.

Because **cvs** was originally designed to perform remote execution using not **ssh** but rather an insecure program called **rsh**, it is necessary to indicate to **cvs** that only **ssh** should be used. This can be accomplished by setting and exporting the `CVS_RSH` environment variable. Using the Bash shell, which is standard with Cygwin, most Linux systems and many Unix systems, you would input the following commands to the command-line interface:

```
CVS_RSH=ssh
export CVS_RSH
```

To avoid having to type these two lines every time you log in to a Linux or Unix shell or start the Cygwin shell window, you should add them to your `.bash_profile` shell initialization file.

Since the server supports several CVS repositories, you must tell CVS which repository you are working with. Although all **cvs** commands allow you to identify the repository with a command-line argument, it is usually more convenient to define an environment variable. The following lines should be added to your `.bash_profile` shell initialization file:

```
       USER=user
CVSHOST=whitechuck.rfpk.washington.edu
CVSPATH=/usr/local/cvs_repos/rfpk/r1
CVSROOT=:ext:$USER@$CVSHOST:$CVSPATH
       export CVSROOT
```

where *user* is your login user name on whitechuck . *Note the r1 at the end of the line which defines CVSPATH*. This indicates that you want to use the r1 repository rather

than r2 or some other repository. Note: the RFPK Software Team uses the r2 repository.

The Command Line Interface

At present, the only interface available for RFPK's Windows users is the command-line interface. Linux users have a choice of the command-line interface and the *Cervisia* GUI. The GUI is really just a wrapper around the command-line interface, hence it is recommended that all users read the following section.

Getting a Working Copy

To make changes to existing documents or to add new documents you first need to get a working copy of the repository or at least the part of the repository that you want to change. You accomplish this with the CVS **checkout** command.

For example, you can checkout the entire r1 repository with the following command:

```
cvs checkout r1
cvs update -P
```

To checkout a particular module, in this case the *muscle* module, you would use this command:

```
cvs checkout muscle
cvs update -P
```

When you issue a **checkout** command, **cvs** inserts a copy of the module in your current directory. In the first example, a directory named `r1` would appear in your current directory. In the second example, a directory named `muscle` would appear.

The **update -P** is not absolutely necessary, but is recommended. The **-P** argument stands for *prune*. The effect is to prune empty directories from your working copy of the source tree. Empty directories are retained forever in the repository because they contain history even though all source code files have been removed.

Your working copy is a directory tree. The root of that tree is named for the repository or the module that you checked out. Within the directories of this tree, you will find the documents that are under version control. You are free to modify these documents, using your favorite text editor.

In each of the directories you will also find a subdirectory named `CVS`, which was placed there by **cvs** as part of the checkout process. *Do not modify the CVS directory or any of its contents.*

Getting Updates

As you work on your modifications, your collaborators may have been revising the documents of the same module you are working on. You can easily get your working copy updated with any changes they have made.

To use the CVS **update** command, your current directory must be one of the directories of your working copy. The command will update files in that directory and in any subdirectories. It will not update files in parent directories. Thus positioned, simply issue the command:

```
cvs update -P
```

If the files you are currently working on have been revised since your last **update** or **checkout** command, the updating process will change the affected files in your working copy by inserting modifications from the repository, and offsetting differences between your working copy and the repository with strings like <<<<<<< or >>>>>>>, depending on whether the difference is in the the working copy or in the repository. When this happens, you need to edit your working copy to resolve any conflicts and remove the <<<<<<< and >>>>>>> strings.

Committing Your Work

When you have reached a point when you would like to apply the modifications that you have made in your working copy to the repository itself, you use the CVS **commit** command. This command is similar to the *check-in* command in some other revision control systems, except that after **commit**, all the files in your working copy are still checked out, and you can immediately resume making additional modifications to them.

Just as with **update** and most other CVS commands, to use the CVS **commit** command your current directory must be one of the directories of your working copy. The command will consider modifications to files in the current directory or in any of its subdirectories. It will not commit files in parent directories.

To commit modifications to files in the current directory and, recursively, any of the subdirectories, use the command:

```
cvs commit
```

If you only want to commit changes to specific files, then list them explicitly in the command:

```
cvs commit file1 file2 ... filen
```

You can provide comments about the purpose of your change by using the `-m` option to provide a revision message:

```
cvs commit -m "Fix for bug 13.224."
```

Revision messages go into a log which becomes part of the revision history in the repository.

If you do not provide the `-m` option, **cvs** will start a text editor for you, so that you can type in the revision message. Depending on what system you are running on, the default editor might be **emacs** or **vi**. This can be confusing, especially if the default editor is one that you are not used to. To avoid getting the default editor, all you have to do is add a pair of lines to your `.bash_profile` shell configuration file. Suppose that you prefer **emacs**:

```
CVSEDITOR=emacs
export CVSEDITOR
```

If someone else has made changes to the repository copy of a file since you checked out or updated your working copy, the **commit** command will report errors and will fail to incorporate your changes in the repository. Here is an example of the output that you might expect to see. A user called *jones* has attempted to commit a file called *Makefile*, and the CVS **commit** command has returned the following output:

```
cvs server: Up-to-date check failed for ‘Makefile’
cvs [server aborted]: correct above errors first!
cvs commit: saving log message in /tmp/cvs610c74f4.1
```

In the above example, **commit** reports a failure in the *up-to-date check* for a particular file, which simply means that a copy of this file has been committed since your working copy was last checked out or updated. To correct this error, proceed as follows:

1. Update your working copy of the file. You could use the following command to update the particular file in question, which in this example is called `Makefile`.

   ```
   cvs update Makefile
   ```

2. Edit the file, using your favorite text editor. Differences between your working copy and the repository will be identified with >>>>>>> and <<<<<<< strings, as explained above, in the section that describes the **update** command.
3. Once you have removed any conflicts, as well as the markers that were inserted by the **update** command, execute the **commit** command again:

   ```
   cvs commit Makefile
   ```

   This time, **commit** should accept your changes, and record a new revision.

What's the Difference?

The CVS **diff** command will display the differences between any files in your working copy and their counterparts in the repository, or between different revisions in the repository. The need to do this arises frequently, for a number of reasons:

- You have made changes and you need a reminder or a record of what the differences are.
- You know, or suspect that someone else has committed changes to a file that you are currently working on, and you would like to know what these are before attempting a **commit** or an **update**.
- You would like to know how a version from the past differs from your working copy, or how two versions from the past differ from each other. This can be especially useful when the version from the past was released to the public.

The simplest form of the **diff** command reports all differences between files in the current directory and, recursively, all subdirectories and their counterparts in the repository:

```
cvs diff
```

Since the differences may be voluminous, you may want to pipe the output into a pager command, such as **more** or **less**:

```
cvs diff | more
```

Another way to handle voluminous output is to store it as a file:

```
cvs diff > file
```

Yet another option is to send the output to a printing agent such as **lpr**:

```
cvs diff | lpr
```

To find the differences between specific files in your working copy and their counterparts in the repository, you enumerate the files in the command:

```
cvs diff file1 file1 ... filen
```

If you are interested in difference between your current working copy and a version released in the past, you use the `-r` option letter to introduce the tag associated with the version:

```
cvs diff -r some-tag
```

To see the differences between two versions, you can specify two `-r` options:

```
cvs diff -r tag1 -r tag2
```

There are several options which affect the format of the output from the **diff** command. The `-b` option ignores the difference in the amount of whitespace between files. The `-B` option ignores differences due to the insertion or deletion of blank lines.

The normal output from the CVS **diff** command simply shows lines that are in one file but not in the other and *vice versa*. It does not show lines present in both files which surround these difference. The `-c` option provides three lines of *context* before and after each set of lines which differ. This can be useful in helping you to determine exactly where the differences are located within their respective files. It is also a necessity, if you want to store the differences to a file for later use with a patch program, such as the Unix **patch** command.

Adding Files and Directories

Suppose that we would like to add to the repository a document similar to the tutorial that you are reading. Suppose further, that within our working copy there is a directory named `howto` which contains a directory for each document of this type,

and that this latter directory contains the individual documents. For example, this tutorial consists of a directory and two files:

```
howto/CVS-rfpk-HOWTO
howto/CVS-rfpk-HOWTO/CVS-rfpk-HOWTO.xml
howto/CVS-rfpk-HOWTO/Makefile
```

Now suppose that the document that we wish to add will be a tutorial about installing SPK. In this example the text of the document will be called `SPKinstall-rfpk-HOWTO.xml`. There will also be a description file for the **make** utility which will be named `Makefile`. In order to keep all files associated with this document together, we will place these two files in a directory that we will name `SPKinstall-rfpk-HOWTO` .

To create this new document, we proceed as follows:

1. Create a directory and the two files using the normal tools installed on our system. For example, under Unix/Linux, or in a Cygwin environment installed under Windows, we could do the following:

   a. Within our working copy, go to the `howto` directory by using the `cd` command. `howto` is the directory in which we will create our new document.

   b. Create the new directory in our working copy, and then have **cvs** add it to the repository:

      ```
      mkdir SPKinstall-rfpk-HOWTO
      cvs add SPKinstall-rfpk-HOWTO
      ```

   c. Next go to the new directory

      ```
      cd SPKinstall-rfpk-HOWTO
      ```

      and use our favorite text editor to create the initial versions of the files `SPKinstall-rfpk-HOWTO.xml` and `Makefile`.

   d. Finally, have **cvs** add the new files to the repository structure and then commit our initial versions:

      ```
      cvs add SPKinstall-rfpk-HOWTO.xml Makefile
      cvs commit
      ```

Removing Files and Directories

Files and directories can be removed using the CVS **remove** command which is very much the reverse of the CVS **add** command. Suppose, for example, we decide that we no longer have a use for the file called `Makefile` that we added to the repository in the example in the section above. We can remove the file as follows:

1. Go to the directory in our working copy which contains the file in question and remove it. For example, under Unix, Linux or Cygwin, we would use the following command to remove the file:

   ```
   rm Makefile
   ```

2. Still in the directory from which the file was removed, have **cvs** complete the work:

```
cvs remove
cvs commit
```

Conclusion (command-line interface)

The commands describe above should be enough to get you started. In the books that are free for downloading via the URLs listed in the introduction, there are additional commands and a great deal of useful information.

Perhaps the most powerful CVS command not covered in this tutorial is the **tag** command, which can be used to place an identification tag on all files which comprise a particular version or release of a product. This tag can subsequently be used to reference that release with the **diff** command, in order to analyze differences between it and another release or between it and the current version. A version tag can be used with **checkout** so that the version can be easily reconstituted. A version tag can also be the basis for creating a separate branch for a released version of a product, in which bug fixes can be developed and tested without interfering with current development.

The Cervisia GUI

Cervisia is a GUI wrapper around the command-line version of CVS. It is part of the support for KDE software development that is now a standard part of the RedHat Linux distribution. You can easily determine whether the package was installed on your system with the following *rpm* query:

```
rpm -qa | grep cervisia
```

If it is not on your system, you can install it from the RedHat distribution cdroms or download it from the web.

Once the cervisia rpm has been installed, you will probably find a launcher for it in the RedHat menus under **Extras**-> **Programming**-> **Cervisia**. You may want to add a launcher for cervisia to your panel by right-clicking the **Cervisia** item in the above menu sequence, then selecting: **Add this launcher to panel**.

If you have read the previous section on the command-line interface, you will find cervisia to be rather self-explanatory. The following subsections should help get you started.

Repository Access

In order for *cervisia* to be able to find the repository, set up your environment variables just as you would if you were using the command-line interface. You will find a full description of this process in the section titled *Environment Variables for Repository Access*, above,

Setting Up Your Sandbox

In order to get a local copy of the repository, using the command-line interface is recommended. This process is covered in the section *Getting a Working Copy*, above,

Opening Your Sandbox

Now start up *cervisia*. Click the item **File**-> **Open Sandbox**. A file access window will appear. Within this window, navigate to the directory with the same name as your repository (eg.`r2`) and double-click. If you now click the **OK** button, you will open the entire repository. If you are not interested in the entire repository, but just a portion farther down in the hierarchy, continue to navigate down to the portion you want to open, double-click on that directory name, then press **OK**.

The next time you open your sandbox, you can select your sandbox from the menu list **File**-> **Recent Sandboxes**.

Cervisia Settings

There are two options in the **File**-> **Settings** menu which are recommended, because they mirror the default behavior of the command-line interface and thus are consistent with most documentation about CVS. The recommended options are **Update Recursively** and **Commit and Remove Recursively**.

CVS Commands

All of the most frequently used commands are available in *cervisia* menus. You typically select a file or directory, then select the command from a menu. The screen is split. The actual output of the CVS command, as you would see it if your were using the command-line interface, is copied to a scrolling window, so that you can monitor the results of your action.

Conclusion (GUI interface section)

The *cervisia* GUI is a useful tool, especially for those who strongly dislike command-line interfaces. It is, however, a wrapper around the command-line version of CVS, and it is impossible to understand without some understanding of the way in which that version works and of its output. Nearly all documentation about CVS describes the behavior of the command-line interface.

Some Server Administration

Most of the administration of a CVS repository can be carried out by ordinary users from their remote systems. This section is reserved for tasks that are best performed by someone who has root access to the server.

Creating A New Repository

To create a repository, you need root privilege on the server.

1. Log in to the server and get root privilege by using the **su** command.
2. Using the **cd** command, go to the directory that will be the root directory of your repository. If the directory does not exist, create it at this time and then go to it.
3. Initialize the repository:

```
cvs -d $(pwd) init
```

4. If you have not done so, create a group of users of this repository. In this example, lets assume that the group name is *dvl*.

   ```
   /usr/sbin/groupadd -r dvl
   ```

   All of the users of this group must be ordinary users of the server. Suppose that the users are tom, dick and harry. Add them to the group as follows

   ```
   /usr/bin/gpasswd -a tom dvl
   /usr/bin/gpasswd -a dick dvl
   /usr/bin/gpasswd -a harry dvl
   ```

   The repository directory and all subdirectories must have the given group (*dvl* in this example) as its group. In addition, the group must have read and write privileges in the directory, and the *setgid bit* must be set so that any directories or files created in the directory will have *dvl* for group: Assume that we are still positioned in the directory:

   ```
   chgrp -R dvl .
   chmod 2775 .
   ```

   If there are any directories under this directory, go through the hierarchy recursively and repeat the above **chmod** command line on them. *Do not* use the recursive *-R* option, because you do not want to turn on the *setgid bit* for ordinary files.

5. Finally, mark the repository as a *module*, so that it can be easily accessed by the CVS *checkout* command. This step does not require root privilege and does not have to be performed while logged in to the server. It is explained in the next section.

Defining Modules

A *module* in CVS is a type of alias for a path within a repository. The repository itself should be defined as a module, so that it can be retrieved by name with the CVS **checkout** command. Within the repository hierarchy, it is often convenient to define subdirectories which are themselves the root directories of projects as modules.

You will perform the following steps as an ordinary user on a client machine. For this to work, however, you must also be installed as a user on the server and must be a member of the repository's group on that machine. Your environment variables must be set up as described above.

1. Checkout a copy of the CVSROOT directory from the root node of the repository, and then go to it:

   ```
   cvs checkout CVSROOT/modules
   cd CVSROOT
   ```

2. Using your favorite editor, edit the `modules` file in this directory. Suppose that the repository is named *myrepos*. You would insert the following line in the file:

   ```
   myrepos .
   ```

   To designate the subdirectory, for example `src/c/project-1` , you would insert the line

   ```
   project-1 src/c/project-1
   ```

Since the module names are aliases, the module name does not have to be the same as the directory name. In the above example, we could have named the module *integrator* as follows:

```
integrator src/c/project-1
```

3. After saving the modules file and leaving your editor, commit the changes and release the your copy of the CVSROOT directory:

```
cvs commit modules
cd ..
cvs release -d CVSROOT
```

Notes

1. http://muir.rfpk.washington.edu:800/SoftwareBooks/cederqvist-1.11.2.html/cvs.html
2. http://muir.rfpk.washington.edu:800/SoftwareBooks/cvsbook.html