Getting Acquainted with CVS

Matt Gray

April 24, 2004

1 Introduction

The Concurrent Versioning System (CVS) is a great way to manage documents, source code, and numerous other file types over a lifetime of edits, allowing multiple people to collaborate without worrying about overwriting someone else's changes. CVS allows for the storage of all documents and files in a central directory, called a *repository*.

The repository holds all of the data for the current and past versions of your files; it has the ability to reconstruct any single version. It is also accessible remotely, via secure means such as SSH. You can revise your files anywhere you have terminal access—you are not tied to a specific computer.

CVS has numerous features, which can be overwhelming at first glance. This paper should serve as an introduction or refresher, as well as a reference. Other sources of information include the `man` pages for CVS (type `man cvs`), the CVS web site (http://www.cvshome.org/), and Google (http://www.google.com/).

2 Your First Repository

This section will take the user through some examples using temporary files in one's home directory. There is absolutely no chance of messing things up, since a sandbox is set up to try CVS techniques without worrying about making changes to an active repository. Users looking for more advanced material may safely skim this section if they are somewhat familiar with CVS.

2.1 What the heck is a repository?

CVS is built around the concept of a *repository*, a central location for all of the files you would like CVS to manage. Without one, the commands are pretty much useless. A repository is simply a fancy term for a directory tree, which can be created anywhere you have write permissions. Once created, a repository is typically a permanent thing, where many different projects are stored. The repository that will be created in the following tutorial is meant to be temporary, although it is a full-fledged repository.

We will create a temporary directory tree to begin learning CVS. This structure will contain both the CVS repository, and some working directories. This document assumes that you are using `csh` or `tcsh` from here on out; the method for changing environment variables, for example, is different in each shell. The same steps can be accomplished with `bash`, but the syntax is different. When such shell-specific commands appear, they will be noted. For APS/Eta users, the default shell is `tcsh`, so the directions can be followed directly.

2.1.1 Directory set-up

For the purposes of this tutorial, we will create some directories underneath your home directory to play around in. Of course, this can be placed anywhere you like, as long as you know the full path to the directory. Here, we will use `~/cvstemp` as our working directory, where '~' is your home directory.

First, let's change to our home directory.

```
[matt@set ~]$ cd
[matt@set ~]$ pwd
/home/matt
[matt@set ~]$
```

Now, let's make a temporary directory to work in:

```
[matt@set ~]$ mkdir cvstemp; cd cvstemp
[matt@set ~/cvstemp]$ pwd
/home/matt/cvstemp
[matt@set ~/cvstemp]$
```

Next, we need to create a directory for the repository to reside in.

```
[matt@set ~/cvstemp]$ mkdir repository
[matt@set ~/cvstemp]$ ls -l
total 4
drwxr-xr-x    2 matt     staff        4096 Feb  5 13:23 repository/
[matt@set ~/cvstemp]$
```

2.1.2 Setting up your shell environment

We're almost ready to run our first `cvs` command. `cvs` needs to know which repository it is operating on every time the command is executed, since many repositories can exist. Typing the full path to the CVS repository is fine for one-time CVS commands (like downloading source code from a remote server) but terrible for daily tasks—a long command line would be cumbersome. Luckily, there is a simple solution.

Every time it is executed, the `cvs` command checks to see if the `CVSROOT` environment variable is defined. When no specific repository is given on the command line, `cvs` will use `CVSROOT` if it exists. This saves lots of typing, as full path names are very tedious to type out. For our example, our repository will be rooted in the directory `~/cvstemp/repository`, or, in the author's case, `/home/matt/cvstemp/repository`.

Commands for altering shell environment variables vary from shell to shell; for `tcsh` or `csh`, the command is `setenv`:

```
[matt@set ~/cvstemp]$ setenv CVSROOT /home/matt/cvstemp/repository
[matt@set ~/cvstemp]$ printenv CVSROOT
/home/matt/cvstemp/repository
[matt@set ~/cvstemp]$
```

Please note that this change applies to only your current shell. For more permanent changes, you will need to modify your `~/.cshrc` or `~/.tcshrc` file to set the variable every time you open a shell. For the sake of this tutorial, it is fine to have the setting confined to just one

shell, since you will likely not use the repository created in these examples again. If you choose to close the shell you use for this tutorial, when resuming the tutorial you **must re-set your `CVSROOT` environment variable**. Also notice that the full path is specified.

2.1.3 Initializing the repository

You are now ready to run your first `cvs` command; `cvs init`. `init` is executed only at the creation of a new repository, and should not be used on existing repositories. All the command does is create a directory called `CVSROOT` in your repository directory. This is used to store administrative information about the repository as a whole. Luckily, we don't need to worry about the internals of how the repository works; CVS manages it for us, once it has been initialized.

To set up the repository for the first time, run the command

```
[matt@set ~/cvstemp]$ cvs init
[matt@set ~/cvstemp]
```

If there are no errors, you will simply return to the prompt. You can verify that the `CVSROOT` directory was created by listing the directory `~/cvstemp/repository`:

```
[matt@set ~/cvstemp]$ ls -l repository
total 4
drwxrwxr-x    3 matt     staff        4096 Feb  5 13:39 CVSROOT/
[matt@set ~/cvstemp]$
```

Notice how it has group write permissions. CVS is designed to allow a group of people work on the same set of files, hence the directory is group writable. If you decide to use CVS for private data, restricting the permissions would be a wise idea.

2.2 Projects, importing, and organization

The repository needs to be initialized only once; it is now ready to be used. We shall call the repository's directory (the one you set to the `CVSROOT` variable) the *CVS root* directory. In order to use CVS effectively, we must understand how it organizes the files it manages.

2.2.1 Organization in CVS

Files in CVS are grouped into what we will refer to as 'projects', but are actually subdirectories (excluding the special administrative directory `CVSROOT`) under the CVS root. These project trees can have many subdirectories underneath the main project directory. A project can be thought of a collection of files—not necessarily a formal 'project'. Since projects are subdirectories under the CVS root, two projects cannot have the same name, since two directories of the same name cannot exist in the same directory.

There are much finer levels of granularity than projects, as a single project may have several distinct versions in parallel; these are advanced concepts beyond the scope of this tutorial.

2.2.2 Starting a new project

All projects begin with a directory. Therefore, you must first organize the files and directories you would like to include in the new project into a top-level directory. Do not worry if you don't have all the files started, or in quite the order you want—it is possible to add, remove, and shift files once they are part of a CVS project. For this example, we will create a small directory tree with some simple text files.

```
[matt@set ~/cvstemp]$ mkdir tutorial
[matt@set ~/cvstemp]$ cd tutorial
[matt@set tutorial]$ cat > sample.txt
This is a sample text file
```

Type CTRL-D to close the file

```
[matt@set tutorial]$ mkdir test
[matt@set tutorial]$ cat > test/sample2.txt
This is another sample text file, in a directory.
```

Type CTRL-D again

```
[matt@set tutorial]$
```

Now, we have a directory `tutorial` containing a text file and a directory, which contains another text file. We're now ready to let CVS manage these files. Now, when importing this directory tree, `tutorial` has no impact on the name of the project; `cvs` always looks at the *current working directory* when it performs operations. Therefore, when it considers what files to import, it will recursively find files under the current directory, `tutorial`. The arguments to `cvs import` determine the actual name of the project. All files and directories *under* the top-level directory (`tutorial` in this case) will retain their current names.

The command to import a directory tree rooted at the current working directory into a CVS repository is as follows:

```
cvs import projectname vendortag releasetag
```

The command breaks down as follows: the '`cvs`' program is executed, and told to perform the '`import`' command. All files and subdirectories under the current working directory will be added to the repository under the project name '`projectname`'. The '`vendortag`' option specifies the name of the main *branch*. Branches in CVS are a complicated matter, and will be discussed in depth in a later section. Finally, the '`releasetag`' parameter gives a *release name* to the files being imported. A release is a way of grouping files of different versions together conceptually. For example, if you were importing a software project, you might import with a `releasetag` of 'release-0_0'. Later, if you wanted to fetch the files you imported at the beginning, you would request them by their `releasetag`. **For now, we can ignore these details.**

A simple method for naming the `vendortag` and `releasetag` is as follows: give your username as the vendortag, and 'start' as the release tag. This way, it is clear that the branch was initiated by you, and the 'start' release contains the files as you first imported them.

The author would use the command line

```
cvs import tutorial matt start
```

to import the project in the `tutorial` directory. Note that we do not *have* to name this project tutorial; the project name is not dependent on the name of our current working directory.

Before you execute your first import command, it is important that your `EDITOR` environment variable points to your text editor of choice. Every time CVS performs a change, it asks that you provide a descriptive message that discusses the changes made. The messages are created by invoking the program specified in `EDITOR`, and saving that file as the message in the CVS log. If you are comfortable with `vi`, it is usually the default editor.

The log message may also be specified on the command line, by following the *cvs command string* with the `-m` option, followed by a string of text. An example of this usage is as follows:

```
cvs import -m 'Initial import' myproject matt start
```

You should now be comfortable with the import command, so let's import our first project. Change to the `tutorial` directory, and execute the following commands:

```
[matt@set tutorial]$ pwd
/home/matt/cvstemp/tutorial
[matt@set tutorial]$ cvs import -m 'Import of some test files' tutorial matt start
N tutorial/sample.txt
cvs import: Importing /home/matt/cvstemp/repository/tutorial/test
N tutorial/test/sample2.txt

No conflicts created by this import

[matt@set tutorial]$
```

The output following the command shows the progress of CVS as it descends through the directory structure. It first encounters the file `sample.txt`, and lists:

```
N tutorial/sample.txt
```

The '`N`' signifies that this file was previously unseen in the repository. This is logical, since we just added the project! Notice that the filename is prefixed by the project's (directory) name. If you had executed `cvs import myproj matt start` instead, it would show

```
N myproj/sample.txt
```

as the name of the imported file, *even though* the directory name is `tutorial`.

Next, CVS reports that it is descending into a subdirectory:

```
cvs import: Importing /home/matt/cvstemp/repository/tutorial/test
```

The file `test/sample2.txt` is similarly listed in the report:

```
N tutorial/test/sample2.txt
```

as it is a new file. The final line of the program's output, '`No conflicts created by this import`', simply means that there wasn't another project or file of the same name in the repository, such that the differences couldn't be resolved.

You have now successfully imported your first project into CVS. Pat yourself on the back.

2.2.3 I've imported—so what?

Now that you have a project imported correctly, after all that work, what do you do with it? After all, the files are still sitting in your `tutorial` directory—what was the point of all this? This is an important question, because understanding the answer is central to understanding the principle behind the CVS method of managing documents. CVS is designed to exercise very fine control over files: enough to allow audit trails and concurrent editing. It can hardly do this when you are free to arbitrarily edit the files in your `tutorial` directory, which you just imported.

CVS isn't magic—it can't keep track of when and how you change those files without some mechanism for discovering these data. Even though you have successfully imported the files, this just gives CVS a snapshot of their state—their contents. If you were to edit the files in `tutorial`, you would have to go through a lot of trouble to get those changes placed into CVS properly.

So, why do this if you can't edit the files? The answer is that you **can** edit the files, but not these particular copies. CVS is a bit like borrowing a book from the library, making annotations, and returning it to the library. The only difference is that since the media is electronic, you can have many identical copies checked out, and then *correlate the compatible changes* on check-in, and *alert the user of conflicts between modified versions*. This is the heart of the CVS concept.

In fact, to start editing within CVS's purview, the best thing to do is to *delete* the directory containing these original imported files! This sounds like lunacy to the careful user, so for now we will simply *rename* the directory `tutorial` to something else, just in case:

```
[matt@set tutorial]$ cd ..
[matt@set ~/cvstemp]$ mv tutorial old-tutorial
[matt@set ~/cvstemp]$ ls
old-tutorial  repository
[matt@set ~/cvstemp]$
```

2.2.4 Checking out

The first step in our library analogy involves checking out our "book". If we want to add to the "book" for future reference, we need to get our hands on it first! As CVS is a virtual electronic "library", we can do this very easily.

The command, unsuprisingly, is `cvs checkout projectname`. On execution, it will download the *latest* version of the project `projectname` into the current working directory. It will be contained within a top-level directory, named the same as the project. You can override this default directory name with the `-d` option—see the detailed section on the CVS commands later in this document.

Since the project that you just checked in is named '`tutorial`', you might guess that the command to retrieve it looks like

```
[matt@set ~/cvstemp]$ cvs checkout tutorial
cvs checkout: Updating tutorial
U tutorial/sample.txt
cvs checkout: Updating tutorial/test
U tutorial/test/sample2.txt
[matt@set ~/cvstemp]$
```

Well, you guessed correctly. Go ahead and execute the command, and let's examine the output. CVS reports that it is '`Updating tutorial`'. As there is no version of `tutorial` checked out in the current working directory, it "updates" it by retrieving the latest version from the CVS repository.

Notice the '`U`' instead of an '`N`' like we saw during the importation; a `U` is short for "update" in CVS messages. As expected, CVS retreives our two text files, as well as the directory `test`. Let's examine our new copy of `tutorial` freshly checked out from our "library"!

```
[matt@set ~/cvstemp]$ cd tutorial
[matt@set tutorial]$ ls -l
total 12
drwxr-xr-x    2 matt     staff        4096 Feb  5 19:31 CVS/
-rw-r--r--    1 matt     staff          27 Feb  5 18:51 sample.txt
drwxr-xr-x    3 matt     staff        4096 Feb  5 19:31 test/
[matt@set tutorial]$ ls -l test
total 8
drwxr-xr-x    2 matt     staff        4096 Feb  5 19:31 CVS/
```

```
-rw-r--r--    1 matt     staff          50 Feb  5 18:51 sample2.txt
[matt@set tutorial]$
```

Everything seems to be in order, except for two mysterious `CVS` directories. These contain administrative information, such as the CVSROOT that this checked out copy of `tutorial` came from. Also notice that the files have their original modification times—importing and checking out did not affect them at all.

Congratulations, you have checked out a project successfully. You're only a step away from actual editing! Have a beer.

2.3 Editing files under CVS

2.3.1 The importance of updates

In order to edit files with CVS, they must first be checked out; we now have a copy of `tutorial` checked out from the repository to work on. Creating new revisions is as simple as modifying the files in your favorite text editor. Before you fire up `vi` or `emacs`, wouldn't it be worthwhile to ensure that the copy in the `tutorial` directory is *really* up to date? After all, while you read this, someone may have made alterations to the files, and now your files are out of date!

For this reason, it is usually a good idea to *update* your working copy before editing, especially if you haven't looked at it for a while (hours, days). For practical purposes, this isn't necessary *now*, since you just recently checked out a copy of `tutorial`. However, updating before starting an editing session is a great habit to get into.

Updates transfer information *from* the repository to your local copy; changes you have made are **not** transferred *to* the repository. The command to perform this update is simply '`cvs update`'. When updating an entire working copy of a project, make sure you are in the top level of that copy's directory tree. Otherwise, CVS will skip all files and directories not contained within the current working directory; CVS only recurses down a directory tree.

Executing the command looks like this:

```
[matt@set tutorial]$ cvs update
cvs update: Updating .
cvs update: Updating test
[matt@set tutorial]$
```

If the command was executed while our current directory was `tutorial/test`, here's what would happen:

```
[matt@set test]$ cvs update: Updating .
[matt@set test]$
```

Notice that the current working directory, '`.`', was updated. The parent directory was not considered for the update.

2.3.2 Local changes

Now, to continue our walkthrough, edit the file `tutorial/sample.txt` to remove the word 'sample', so the file reads as:

```
[matt@set tutorial]$ cat sample.txt
This is a text file
[matt@set tutprial]$
```

If you execute a `cvs update` from the `tutorial` directory, notice how the output changes:

```
[matt@set tutorial]$ cvs update
cvs update: Updating .
M sample.txt
cvs update: Updating test
[matt@set tutorial]$
```

There is a new status letter, 'M', indicating that this file has been locally modified—it differs from what is currently in the repository. Remember, `cvs update` only propagates changes *from* the server *to* your copy; therefore, your local copy of `sample.txt` still has your changes. For now, we will not consider the possibility that new information from the server conflicts with the changes you have made locally; this will be addressed in a later section.

2.3.3 Making local changes global

We've seen how to move changes from the server to our working copy; the next step is to propagate changes from our working copy back to the repository. The way to do that is to *commit* them.

As you might guess, the command to do this is `cvs commit`:

```
[matt@set tutorial]$ cvs commit
```

After executing the command, the text editor defined in your `EDITOR` environment variable (usually vi) will open, with the following file:

```
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with 'CVS:' are removed automatically
CVS:
CVS: Committing in .
CVS:
CVS: Modified Files:
CVS:    sample.txt
CVS: ----------------------------------------------------------------------
```

CVS allows you to record a descriptive message every time you modify a set of files. This provides an annotated audit trail for your documents. It is important to write precise comments describing your changes. "Changed some stuff" or "Major changes" are examples of poor comments. For this example, "Re-worded sentence", or "Wording change" would be more appropriate. Three months from now, a log message like "Stuff" will have no meaning whatsoever. Worse, it will be meaningless to others working on the same document. Please note that you should *not* include details like line numbers, or the text that was deleted; CVS tracks these for you.

All of the lines beginning with `CVS:` are treated as comments, and are not saved. Type a message that describes what you did, save the file, and exit your editor. CVS will then resume, and complete the operation:

```
cvs commit: Examining .
cvs commit: Examining test
Checking in sample.txt;
/home/matt/cvstemp/repository/tutorial/sample.txt,v  <--  sample.txt
new revision: 1.2; previous revision: 1.1
```

```
done
[matt@set tutorial]$
```

Notice that sample.txt received a new revision. Even though you made a change to the file, CVS knows about *both* versions of the file! CVS also handles automatic version numbering, which we will discuss in further detail later.

Now that the changes have been committed, `cvs update` will no longer report `sample.txt` is '`M`', or modified:

```
[matt@set tutorial]$ cvs update
cvs update: Updating .
cvs update: Updating test
[matt@set tutorial]$
```

2.3.4 Adding new files

Projects have a way of acquiring more and more files, so we require a way to add files to a given project. In order to do this, we require an up-to-date working copy of the project. We already have a working copy checked out—it is in the `tutorial` directory. It is relatively up-to-date, as we just executed a `cvs update`.

To add a file, simply place the desired file where you like in the project's directory tree. For example, let's add a sample3.txt to the `tutorial/test` directory:

```
[matt@set tutorial]$ cd test
[matt@set test]$ cat > sample3.txt
I forgot to add this sample text file!
```

Type CTRL-D

```
[matt@set test]$ cd ..
[matt@set tutorial]$
```

The file now exists in our working copy, but if we run yet another `cvs update`, we discover a new line in the output:

```
[matt@set tutorial]$ cvs update
cvs update: Updating .
cvs update: Updating test
? test/sample3.txt
[matt@set tutorial]$
```

The file `tutorial/test/sample3.txt` appears in the output, with the new status character '`?`', which means that CVS doesn't have the file in its repository at all. The fact that CVS does not automatically add such unknown files is useful. For example, during the compilation of a large program, many intermediate files are created that are not worth saving—for example, the `.o` files. CVS will ignore these files if they are not explicitly added to the repository.

Adding files is accomplished with the `cvs add` command:

```
[matt@set tutorial]$ cvs add test/sample3.txt
cvs add: scheduling file 'test/sample3.txt' for addition
cvs add: use 'cvs commit' to add this file permanently
[matt@set tutorial]$
```

According to the output, we must `commit` the new file, as well. However, before we execute a `cvs commit` command, let's examine the output of a `cvs update` after we have `add`ed the file:

```
[matt@set tutorial]$ cvs update
cvs update: Updating .
cvs update: Updating test
A test/sample3.txt
[matt@set tutorial]$
```

You guessed it, the status letter '`A`' means that the file `tutorial/test/sample3.txt` will be *added* to the repository after `cvs commit` is run:

```
[matt@set tutorial]$ cvs commit -m 'Added file'
cvs commit: Examining .
cvs commit: Examining test
RCS file: /home/matt/cvstemp/repository/tutorial/test/sample3.txt,v
done
Checking in test/sample3.txt;
/home/matt/cvstemp/repository/tutorial/test/sample3.txt,v  <--  sample3.txt
initial revision: 1.1
done
[matt@set tutorial]$
```

Note the use of the option `-m 'Added file'`, which uses the message "Added file" in the change log, rather than opening `vi` to enter the message. Also, notice that `sample3.txt` is placed into a file in our *repository*, not our working directory. This is CVS *checking in* a new "page" of our "library book". If someone else were to check out or update a a copy of the `tutorial` project, the new file `test/sample3.txt` would be downloaded and added to their working copy.

Once the commit operation is complete, the new file has been added to the project—an update will not list it as either an unknown or added file.

2.3.5 Removing files

In a perfect world, deletions would not be necessary. Unfortunately, such is not the case. Throughout the lifetime of a project, it is sometimes necessary to remove files; naturally, CVS provides a mechanism to do this.

The command to remove a file is '`cvs remove`'; however, the actual file itself must be removed from your working copy first. If we attempt to remove `tutorial\sample.txt` from CVS, we get the following error:

```
[matt@set tutorial]$ cvs remove sample.txt
cvs remove: file `sample.txt' still in working directory
cvs remove: 1 file exists; remove it first
[matt@set tutorial]$
```

The solution is to run `cvs remove` after deleting the file:

```
[matt@set tutorial]$ rm sample.txt
[matt@set tutorial]$ cvs remove sample.txt
cvs remove: scheduling `sample.txt' for removal
```

```
cvs remove: use 'cvs commit' to remove this file permanently
[matt@set tutorial]$ cvs commit -m 'File removal'
cvs commit: Examining .
cvs commit: Examining test
Removing sample.txt;
/home/matt/cvstemp/repository/tutorial/sample.txt,v  <--  sample.txt
new revision: delete; previous revision: 1.2
done
[matt@set tutorial]$
```

`sample.txt` is really gone, now—or is it? CVS never really deletes a file; it merely removes it from the active version of the project. If we decide later that `sample.txt` was important, it is retrievable. To recover a file that was `cvs remove`'ed but not committed, simply use the `cvs add` command on the file name. If the change is already committed, use the `cvs update` command, *specifying the file name.*

In the example above, we committed the change. Bringing the file back is slightly more complicated. For now, follow these steps without worrying about the mechanics; they will be explained later.

```
[matt@set tutorial]$ cvs update -j 1.3 -j 1.2 sample.txt
U sample.txt
[matt@set tutorial]$ cvs update
cvs update: Updating .
A sample.txt
cvs update: Updating test
[matt@set tutorial]$ cvs commit -m 'added sample.txt back in'
cvs commit: Examining .
cvs commit: Examining test
Checking in sample.txt;
/home/matt/cvstemp/repository/tutorial/sample.txt,v  <--  sample.txt
new revision: 1.4; previous revision: 1.3
done
[matt@set tutorial]$
```

2.3.6 Moving / Renaming files

CVS has no provision to move or rename a file; one must move the file, `cvs remove` the old file by referring to its *old location*, and then `cvs add` the file in its new location. CVS will not link the history of the two files; it is a good idea to note in the log of the new file its old location, so that its previous log may be accessed!

For this reason, moving and renaming files should be avoided if possible.

2.3.7 Directories

We've discussed how to add, remove, and rename files. Directories use the same CVS commands, with a few subtleties. Directories may only be removed if they don't contain any files—that is, there are no files in the current project version. Another difference is that changes to the directory structure do not need to be committed.

2.4 File information

CVS maintains a wealth of information about each file in a given project; this section will examine how to view and interpret it.

2.4.1 The `status` command

The main command to view information about a file is the '`cvs status`' command. Let's look at the status of `tutorial/sample.txt`:

```
[matt@set tutorial]$ cvs status sample.txt
===================================================================
File: sample.txt        Status: Up-to-date

   Working revision:    1.4     Fri Feb  6 22:31:31 2004
   Repository revision: 1.4     /home/matt/cvstemp/repository/tutorial/sample.txt,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

[matt@set tutorial]$
```

There is a lot of information here; first, a the file name and a brief description of its current status is displayed. If you encountered a strange letter in the output of `cvs update`, you could run `cvs status` on that file for an explanation of what the letter means. Also, two version numbers are present: the version present in the working directory, the "Working revision", and the latest version number in the repository. The final three parameters are for slightly more advanced operations; they will be discussed later.

2.4.2 The `log` command

Another useful command is the '`cvs log`' command. It will display the log associated with a particular file, or, if given no other arguments, it will recurse and display the logs for all files in the current working version.

```
[matt@set tutorial]$ cvs log sample.txt

RCS file: /home/matt/cvstemp/repository/tutorial/sample.txt,v
Working file: sample.txt
head: 1.4
branch:
locks: strict
access list:
symbolic names:
        start: 1.1.1.1
        matt: 1.1.1
keyword substitution: kv
total revisions: 5;     selected revisions: 5
description:
----------------------------
```

```
revision 1.4
date: 2004/02/06 23:24:34;  author: matt;  state: Exp;  lines: +0 -0
added sample.txt back in
----------------------------
revision 1.3
date: 2004/02/06 22:31:31;  author: matt;  state: dead;  lines: +0 -0
File removal
----------------------------
revision 1.2
date: 2004/02/06 18:33:03;  author: matt;  state: Exp;  lines: +1 -1
Changed wording.
----------------------------
revision 1.1
date: 2004/02/06 00:51:53;  author: matt;  state: Exp;
branches:  1.1.1;
Initial revision
----------------------------
revision 1.1.1.1
date: 2004/02/06 00:51:53;  author: matt;  state: Exp;  lines: +0 -0
Import of some test files
=============================================================================
```

For now, you don't need to understand every element of the `log` listing—just be aware that it is there, and is a nice way to look at the revision history of a particular document.

2.5 Conflicts

Conflicts are inevitable if multiple people are working on the same file. For example, let's say the sentence in `tutorial/test/sample2.txt` needs to be re-worded. We will simulate the effect of two people working on the same file, and demonstrate how to resolve a conflict.

CVS operates on a line-by-line basis; it will silently incorporate two different edits of the same file version *so long as there are no conflicts on a given line.*

2.5.1 Creating a conflict

In order to create a conflict, we need another working copy of the project to alter, so we can alter and commit two different revisions of the same file version. To do this, change back up to your `~/cvstemp` directory:

```
[matt@set tutorials]$ cd ..
[matt@set ~/cvstemp]$
```

We will perform another `cvs checkout`, only adding the `-d` option (mentioned in 2.2.4), so we can control the name of the new directory. This is necessary, as we already have a directory called `tutorial` in our current working directory!

```
[matt@set ~/cvstemp]$ cvs checkout -d tutorial2 tutorial
cvs checkout: Updating tutorial2
cvs checkout: Updating tutorial2/test
U tutorial2/test/sample2.txt
```

```
U tutorial2/test/sample3.txt
[matt@set ~/cvstemp]$
```

Now there are two copies of `tutorial` checked out from the repository, although in different places. Let's see what happens when we make a conflict! Use a text editor to edit the files `~/cvstemp/tutorial/test/sample2.txt` and `~/cvstemp/tutorial2/test/sample2.txt`. Edit the copy in `tutorial` to read:

```
This is another sample text file, in a directory called test.
```

Change the file in `tutorial2` so it is different:

```
This will create a conflict.
```

Now, we will first `commit` in `tutorial`, then in `tutorial2`. Observe what happens:

```
[matt@set ~/cvstemp]$ cd tutorial
[matt@set tutorial]$ cvs commit -m 'Creating a conflict'
cvs commit: Examining .
cvs commit: Examining test
Checking in test/sample2.txt;
/home/matt/cvstemp/repository/tutorial/test/sample2.txt,v  <--  sample2.txt
new revision: 1.2; previous revision: 1.1
done
[matt@set tutorial]$ cd ..
[matt@set ~/cvstemp]$ cd tutorial2
[matt@set tutorial2]$ cvs commit -m 'Creating a conflict, 2'
cvs commit: Examining .
cvs commit: Examining test
cvs commit: Up-to-date check failed for `test/sample2.txt'
cvs [commit aborted]: correct above errors first!
[matt@set tutorial2]$
```

From the error message, it should be clear that we need to run `cvs update`:

```
[matt@set tutorial2]$ cvs update
cvs update: Updating .
U sample.txt
cvs update: Updating test
RCS file: /home/matt/cvstemp/repository/tutorial/test/sample2.txt,v
retrieving revision 1.1.1.1
retrieving revision 1.2
Merging differences between 1.1.1.1 and 1.2 into sample2.txt
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in test/sample2.txt
C test/sample2.txt
[matt@set tutorial2]$
```

Though you had seen *all* of the one-letter status codes, huh? '`C`' is short for conflict. The problem here is that in both tutorial directories, the revision `1.1.1.1` was checked out; one set of changes was committed in `tutorial` with no errors. When trying to commit from `tutorial2`, there was

an error because a newer version existed in the repository; therefore, CVS will not let you `commit` until you `update`. On `update`, however, CVS discovered that you changed the same line as was changed in the previous version.

CVS has no way of knowing which to prefer, and the imaginary user that committed `tutorial` thinks that the file is stored, safe and sound. The only way to solve this problem is for CVS to tell the user working in `tutorial2` about it, and let *them* choose what to do. Open the file `test/sample2.txt` in a text editor, and you will see the following:

```
<<<<<<< sample2.txt
This will create a conflict.
=======
This is another sample text file, in a directory called test.
>>>>>>> 1.2
```

CVS clearly labels the conflict line with several '`<`'s. The top line is from the version in the current directory, and the second line is the version from the repository. They are separated by '`======`'. To resolve this conflict, simply choose a version to keep, and delete the rest of the extra lines. Modify the file so it looks like this:

```
This is another sample text file, in a directory called test, that created a conflict.
```

Now, `commit` this change. Now, if you go back to the `tutorial` directory and run a `cvs update` command, notice that the changes are pulled from the server, but there is no conflict:

```
[matt@set tutorial2]$ cd ../tutorial
[matt@set tutorial]$ cvs update
cvs update: Updating .
cvs update: Updating test
U test/sample2.txt
[matt@set tutorial]$
```

Notice here that the file was merely updated. This is because the *local version* in `tutorial` was unchanged *on that line* with respect to the new changes sent to the repository. CVS simply updated that line of the file. If you look at the file, you'll see that it matches the changes sent from the directory `tutorial2`. In this way, multiple people can work on files at the same time.

3 Quick Reference

Thus far, we have glossed over the basic CVS commands through instructive examples. Here is a summary of the commands covered. Note that the `man cvs` command will provide a comprehensive listing, as well as documentation available on-line.

3.1 Starting a new project—`cvs import`

```
cvs import [-m 'Message'] projectname vendortag releasetag
```

The `import` command creates a new project-level entry in the CVS repository from the files under the current working directory. Import only makes sense when you wish to create a new top-level directory in the CVS tree, not when you'd like to add on to an existing subdivision.

`-m 'Message'` Specify a log message on the command line instead of entering one when CVS invokes your `EDITOR` program.

`projectname` Literally, the name of the 'project'. Corresponds to the top-level directory in the CVS repository. Two projects may not have identical names. This determines the name of the new directory tree in CVS; the name of the current working directory has no effect.

`vendortag` A one-time option specifying roughly 'who' is responsible for this *branch*. A good choice is your username.

`releasetag` Assigns a conceptual name or class to the group of files you are importing. Since an import is the beginning of a project, something like `initial` or `start` are good choices.

3.2 Downloading an existing project—`cvs checkout`

```
cvs checkout [-d directory] projectname
```

The `checkout` command will download a working copy from the CVS repository for you to edit locally. The command can be abbreviated as `co`. The `checkout` command will create a new directory under your current working directory, named for the project. Alternatively, you can specify a directory name for the top-level with the `-d` switch.

`-d directory` Specify a name for the top-level directory the project will be created in. The directory will be in your current working directory.

`projectname` The name of the CVS project you would like to work on. This is the name assigned to the project at import.

3.3 Downloading incremental updates—`cvs update`

```
cvs update
```

The `update` command will bring new changes *from* the server *to* your working copy. It is a good idea to perform an `update` to freshen a working copy before making large-scale changes, in order to minimize potential conflicts. Since it is common to check out a project once, and leave it in your home directory, `update` will download the latest changes others have made without forcing you to completely check out the project again.

`update` is also a good command to use when you'd like to see what will happen on a `commit`. Running `update` from the top-level will show modified, updated, added and removed files, as well as notifying you of any directories or files that are *not* part of CVS, that may have been overlooked.

4 Eta Carinae Web Editing Quickstart

4.1 Introduction

This quick start walk-through assumes that you have some familiarity with CVS. This can be obtained by skimming through the previous sections, or referring back as you read through the following section. Many site-specfic settings are required for this to work properly; the following guidelines should work for any user of the Eta/APS systems.

4.2 Set-up

As was discussed previously, some environment variables need to be set up in order to use CVS properly (unless you want to specify the repository on every CVS command line!).

First, edit your .cshrc file. The file is located in your home directory. Add the following lines at the end of the file:

```
setenv CVSROOT :ext:username@isis.spa.umn.edu:/cvs
setenv CVS_RSH /usr/bin/ssh
```

Please replace '`username`' with your login name on the APS/Eta systems. For the author, this is '`matt`'.

Save and close the file. You will need to tell your shell to re-read your `.cshrc` file this once; every subsequent X terminal you open will automatically read `.cshrc`. This is accomplished by running the command:

```
[matt@isis ~]$ source ~/.cshrc
```

Finally, you will probably want to start a new directory so you can keep all of your CVS-managed files separate from your home directory. I suggest creating a directory called '`cvs`' off your home directory.

4.3 Checking out

Luckily, the Eta Carinae website is already an established project, so you simply need to check it out to work on it. Change your current working directory to your `cvs` directory, and perform the checkout command:

```
[matt@isis ~/cvs]$ cvs checkout etawww
cvs server: Updating etawww
U etawww/eta.css
U etawww/favicon.ico
U etawww/index.html
U etawww/nav.html
U etawww/template.html
cvs server: Updating etawww/archive
U etawww/archive/index.html
U etawww/archive/nav.html
cvs server: Updating etawww/archive/docs
cvs server: Updating etawww/archive/docs/images
cvs server: Updating etawww/archive/images
cvs server: Updating etawww/archive/images/docs
cvs server: Updating etawww/archive/spectra
U etawww/archive/spectra/data.xml
U etawww/archive/spectra/index.html
U etawww/archive/spectra/timeline.fla
U etawww/archive/spectra/timeline.swf
cvs server: Updating etawww/archive/spectra/docs
cvs server: Updating etawww/archive/welcome
U etawww/archive/welcome/index.html
```

```
cvs server: Updating etawww/data
cvs server: Updating etawww/data/docs
cvs server: Updating etawww/data/docs/images

[output continues]

U etawww/treasury/team/index.html
cvs server: Updating etawww/treasury/templates
U etawww/treasury/templates/index.html
[matt@isis ~/cvs]$
```

Perform an `ls`—you should see the directory `etawww` in your current directory. Change into it, and you may begin working!

4.4 Edits

You may now edit files as you see fit. Please remember that even if you inadvertantly 'break' some part of the site, it is easily fixable, since CVS will remember exactly what has been done, and therefore it can be undone trivially.

Remember to use `cvs update` and `cvs commit` properly; for an explanation of how these commands work, simply refer to the tutorial at the beginning of this document.

Also, please provide descriptive log messages. It is also helpful to change only what needs to be changed, so it is clear what the differences between versions are. If you have questions as to why something is implemented the way it is, please contact the author.

5 Work in progress

This document is a work in progress—you should have a good idea of how CVS works. In a later revision of this document, you will learn how to configure CVS to use the APS/Eta CVS server, as well as a handy list of commands and options for quick reference.