Introduction to CVS

Notes For Day 1 of a Two-Day CVS Course

Jim Blandy

Introduction to CVS

These are the notes for Cyclic Software's two-day CVS class. The CVS Versioning System is a program for maintaining and consulting the history of a software project, or any other directory tree of text files.

1 What is CVS for?

CVS maintains a history of a source tree, in terms of a series of changes. It stamps each change with the time it was made and the user name of the person who made it. Usually, the person provides a bit of text describing why they made the change as well. Given that information, CVS can help developers answer questions like:

- Who made a given change?
- When did they make it?
- Why did they make it?
- What other changes did they make at the same time?

2 How to use CVS — First Sketch

Before discussing too many vague terms and concepts, let's look over the essential CVS commands.

2.1 Setting your repository

CVS records everyone's changes to a given project in a directory tree called a *repository*. Before you can use CVS, you need to set the `CVSROOT` environment variable to the repository's path. Whoever is in charge of your project's configuration management will know what this is; perhaps they've made a global definition for `CVSROOT` somewhere.

In any case, on our system, the CVS repository is '`/u/src/master`'. In that case, you would need to enter the commands

```
setenv CVSROOT /u/src/master
```

if your shell is `csh` or one of its descendents, or

```
CVSROOT=/u/src/master
export CVSROOT
```

if your shell is Bash or some other Bourne shell variant.

If you forget to do this, CVS will complain when you try to use it:

```
$ cvs checkout httpc
cvs checkout: No CVSROOT specified!  Please use the ‘-d’ option
cvs [checkout aborted]: or set the CVSROOT environment variable.
$
```

2.2 Checking out a working directory

CVS doesn't work on ordinary directory trees; you need to work within a directory that CVS created for you. Just as you check out a book from a library before taking it home to read it, you use the `cvs checkout` command to get a directory tree from CVS before working on it. For example, suppose you are working on a project named `httpc`, a trivial HTTP client:

```
$ cd
$ cvs checkout httpc
cvs checkout: Updating httpc
U httpc/.cvsignore
U httpc/Makefile
U httpc/httpc.c
U httpc/poll-server
$
```

The command `cvs checkout httpc` means, "Check out the source tree called `httpc` from the repository specified by the `CVSROOT` environment variable."

CVS puts the tree in a subdirectory named '`httpc`':

```
$ cd httpc
$ ls -l
total 8
```

```
drwxr-xr-x  2 jimb          512 Oct 31 11:04 CVS
-rw-r--r--  1 jimb           89 Oct 31 10:42 Makefile
-rw-r--r--  1 jimb         4432 Oct 31 10:45 httpc.c
-rwxr-xr-x  1 jimb          460 Oct 30 10:21 poll-server
```

Most of these files are your working copies of the `httpc` sources. However, the subdirectory called '`CVS`' (at the top) is different. CVS uses it to record extra information about each of the files in that directory, to help it determine what changes you've made since you checked it out.

2.3 Making changes to files

Once CVS has created a working directory tree, you can edit, compile and test the files it contains in the usual way — they're just files.

For example, suppose we try compiling the package we just checked out:

```
$ make
gcc -g -Wall  -lnsl -lsocket  httpc.c   -o httpc
httpc.c: In function `tcp_connection':
httpc.c:48: warning: passing arg 2 of `connect' from incompatible pointer type
$
```

It seems that '`httpc.c`' hasn't been ported to this operating system yet. We need to cast one of the arguments to `connect`. To fix that, line 48 must change from this:

```
    if (connect (sock, &name, sizeof (name)) >= 0)
```

to this:

```
    if (connect (sock, (struct sockaddr *) &name, sizeof (name)) >= 0)
```

Now it should compile:

```
$ make
gcc -g -Wall  -lnsl -lsocket  httpc.c   -o httpc
$ httpc GET http://www.cyclic.com
... HTML text for Cyclic Software's home page follows ...
$
```

2.4 Merging your changes

Since each developer uses their own working directory, the changes you make to your working directory aren't automatically visible to the other developers on your team. CVS doesn't publish your changes until you're ready. When you're done testing your changes, you must *commit* them to the repository to make them available to the rest of the group. We'll describe the `cvs commit` command below.

However, what if another developer has changed the same files you have, or the same lines? Whose changes should prevail? It's generally impossible to answer this question automatically; CVS certainly isn't competent to make that judgment.

Thus, before you can commit your changes, CVS requires your sources to be in sync with any changes committed by the other team members. The `cvs update` command takes care of this:

```
$ cvs update
cvs update: Updating .
U Makefile
RCS file: /u/src/master/httpc/httpc.c,v
retrieving revision 1.6
retrieving revision 1.7
Merging differences between 1.6 and 1.7 into httpc.c
M httpc.c
$
```

Let's look at this line-by-line:

`U Makefile`
: A line of the form '`U` *file*' means that *file* was simply *Updated*; someone else had made a change to the file, and CVS copied the modified file into your home directory.

`RCS file:` ...
`retrieving revision 1.6`
`retrieving revision 1.7`
`Merging differences between 1.6 and 1.7 into httpc.c`
: These messages indicate that someone else has changed '`httpc.c`'; CVS merged their changes with yours, and did not find any textual conflicts. The numbers '`1.6`' and `1.7` are *revision numbers*, used to identify a specific point in a file's history.

  Note that CVS merges changes into your working copy only; the repository and the other developers' working directories are left undisturbed. It is up to you to test the merged text, and make sure it's valid.

`M httpc.c` A line of the form '`M` *file*' means that *file* has been *Modified* by you, and contains changes that are not yet visible to the other developers. These are changes you need to commit.

In this case, '`httpc.c`' now contains both your modifications and those of the other user.

Since CVS has merged someone else's changes into your source, it's best to make sure things still work:

```
$ make
gcc -g -Wall -Wmissing-prototypes  -lnsl -lsocket  httpc.c   -o httpc
$ httpc GET http://www.cyclic.com
... HTML text for Cyclic Software's home page follows ...
$
```

It seems to still work.

2.5 Committing your changes

Now that you have brought your sources up to date with the rest of the group and tested them, you are ready to commit your changes to the repository and make them visible to the rest of the group. The only file you've modified is '`httpc.c`', but it's always safe to run `cvs update` to get a list of the modified files from CVS:

```
$ cvs update
cvs update: Updating .
M httpc.c
$
```

As expected, the only file CVS mentions is '`httpc.c`'; it says it contains changes which you have not yet committed. You can commit them like so:

```
$ cvs commit httpc.c
```

At this point, CVS will start up your favorite editor and prompt you for a log message describing the change. When you exit the editor, CVS will commit your change:

```
Checking in httpc.c;
/u/src/master/httpc/httpc.c,v  <--  httpc.c
```

```
new revision: 1.8; previous revision: 1.7
$
```

Now that you have committed your changes, they are visible to the rest of the group. When another developer runs `cvs update`, CVS will merge your changes to '`httpc.c`' into their working directory.

2.6 Examining changes

At this point, you might well be curious what changes the other developer made to '`httpc.c`'. To look at the log entries for a given file, you can use the `cvs log` command:

```
$ cvs log httpc.c

RCS file: /u/src/master/httpc/httpc.c,v
Working file: httpc.c
head: 1.8
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 8;     selected revisions: 8
description:
The one and only source file for the trivial HTTP client
----------------------------
revision 1.8
date: 1996/10/31 20:11:14;  author: jimb;  state: Exp;  lines: +1 -1
(tcp_connection): Cast address structure when calling connect.
----------------------------
revision 1.7
date: 1996/10/31 19:18:45;  author: fred;  state: Exp;  lines: +6 -2
(match_header): Make this test case-insensitive.
----------------------------
revision 1.6
date: 1996/10/31 19:15:23;  author: jimb;  state: Exp;  lines: +2 -6
...
$
```

Most of the text here you can ignore; the portion to look at carefully is the series of log entries after the first line of hyphens. The log entries appear in reverse chronological order, under the assumption that more recent changes are usually more interesting. Each entry describes one change to the file, and may be parsed as follows:

'`revision 1.8`'
: Each version of a file has a unique *revision number*. Revision numbers look like '`1.1`', '`1.2`', '`1.3.2.2`' or even '`1.3.2.2.4.5`'. By default revision 1.1 is the first revision of a file. Each successive revision is given a new number by increasing the rightmost number by one.

'`date: 1996/10/31 20:11:14; author: jimb; ...`'
: This line gives the date of the change, and the username of the person who committed it; the remainder of the line is not very interesting.

'`(tcp_connection): Cast ...`'
: This (pretty obviously) is the log entry describing that change.

The `cvs log` command can select log entries by date range, or by revision number; see the manual for details on this.

If you would actually like to see the change in question, you can use the `cvs diff` command. For example, if you would like to see the changes Fred committed as revision 1.7, you can use the following command:

```
$ cvs diff -c -r 1.6 -r 1.7 httpc.c
```

Before we look at the output from this command, let's look at what the various parts mean:

`-c`
: This requests that `cvs diff` use a more human-readable format for its output. (I'm not sure why it isn't the default.)

`-r 1.6 -r 1.7`
: This tells CVS to display the changes needed to turn revision 1.6 of httpc.c into revision 1.7. You can request a wider range of revisions if you like; for example, `-r 1.6 -r 1.8` would display both fred's changes and your most recent change. (You can even request that changes be displayed backwards — as if they were being undone — by specifying the revisions backwards: `-r 1.7 -r 1.6`. This sounds odd, but it is useful sometimes.)

`httpc.c`
: This is the name of the file to inspect. If you don't give it specific files to report on, CVS will produce a report for the entire directory.

Here is the output from the command:

```
Index: httpc.c
===================================================================
RCS file: /u/src/master/httpc/httpc.c,v
```

```
retrieving revision 1.6
retrieving revision 1.7
diff -c -r1.6 -r1.7
*** httpc.c     1996/10/31 19:15:23     1.6
--- httpc.c     1996/10/31 19:18:45     1.7
***************
*** 62,68 ****
  }


! /* Return non-zero iff HEADER is a prefix of TEXT.  HEADER should be
     null-terminated; LEN is the length of TEXT.  */
  static int
  match_header (char *header, char *text, size_t len)
--- 62,69 ----
  }


! /* Return non-zero iff HEADER is a prefix of TEXT, ignoring
!    differences in case.  HEADER should be lower-case, and
     null-terminated; LEN is the length of TEXT.  */
  static int
  match_header (char *header, char *text, size_t len)
***************
*** 76,81 ****
--- 77,84 ----
    for (i = 0; i < header_len; i++)
      {
        char t = text[i];
+       if ('A' <= t && t <= 'Z')
+         t += 'a' - 'A';
        if (header[i] != t)
          return 0;
      }
$
```

This output takes a bit of effort to get used to, but it is definitely worth understanding.

The interesting portion starts with the first two lines beginning with `***` and `---`; those describe the older and newer files compared. The remainder consists of two *hunks*, each of which starts with a line of asterisks. Here is the first hunk:

```
***************
*** 62,68 ****
  }


! /* Return non-zero iff HEADER is a prefix of TEXT.  HEADER should be
     null-terminated; LEN is the length of TEXT.  */
  static int
  match_header (char *header, char *text, size_t len)
--- 62,69 ----
  }


! /* Return non-zero iff HEADER is a prefix of TEXT, ignoring
!    differences in case.  HEADER should be lower-case, and
     null-terminated; LEN is the length of TEXT.  */
  static int
  match_header (char *header, char *text, size_t len)
```

Text from the older version appears after the `*** 62,68 ***` line; text from the new appears after the `--- 62,69 ---` line. The pair of numbers in each indicates the range of lines shown. CVS provides context around the change, and marks the actual lines affected with '`!`' characters. Thus, one can see that the single line in the top half was replaced with the two lines in the bottom half.

Here is the second hunk:

```
***************
*** 76,81 ****
--- 77,84 ----
    for (i = 0; i < header_len; i++)
      {
        char t = text[i];
+       if ('A' <= t && t <= 'Z')
+         t += 'a' - 'A';
        if (header[i] != t)
        return 0;
      }
```

This hunk describes the insertion of two lines, marked with '`+`' characters. CVS omits the old text in this case, because it would be redundant. CVS uses a similar hunk format to describe deletions.

Like the Unix `diff` command, output from `cvs diff` is usually called a *patch*, because developers have traditionally used the format to distribute bug fixes or small new features. While reasonably readable to humans, a patch contains enough information for a program to apply the changes it describes to an unmodified text file. In fact, the Unix `patch` command does exactly this, given a patch as input.

2.7 Adding and deleting files

CVS treats file creation and deletion like other changes, recording such events in the files' histories. One way to look at this is to say that CVS records the history of directories as well as the files they contain.

CVS doesn't assume that newly created files should be placed under its control; this would do the wrong thing in many circumstances. For example, one needn't record changes to object files and executables, since their contents can always be recreated from the source files (one hopes). Instead, if you create a new file, `cvs update` will flag it with a '`?`' character until you tell CVS what you intend to do with it.

To add a file to a project, you must first create the file, and then use the `cvs add` command to mark it for addition. Then, the next call to `cvs commit` will add the file to the repository. For example, here's how you might add a README file to the httpc project:

```
$ ls
CVS          Makefile     httpc.c      poll-server
$ vi README
... enter a description of httpc ...
$ ls
CVS          Makefile     README       httpc.c      poll-server
$ cvs update
cvs update: Updating .
? README — CVS doesn't know about this file yet.
$ cvs add README
cvs add: scheduling file `README' for addition
cvs add: use 'cvs commit' to add this file permanently
$ cvs update — Now what does CVS think?
cvs update: Updating .
A README — The file is marked for addition.
$ cvs commit README
... CVS prompts you for a log entry ...
RCS file: /u/jimb/cvs-class/rep/httpc/README,v
done
Checking in README;
/u/src/master/httpc/README,v  <--  README
initial revision: 1.1
```

```
done
$
```

CVS treats deleted files similarly. If you delete a file and then run `cvs update`, CVS doesn't assume you intend the file for deletion. Instead, it does something benign — it recreates the file with its last recorded contents, and flags it with a '`U`' character, as for any other update. (This means that if you want to undo the changes you've made to a file in your working directory, you can simply delete the files, and then let `cvs update` recreate them.)

To remove a file from a project, you must first delete the file, and then use the `cvs rm` command to mark it for deletion. Then, the next call to `cvs commit` will delete the file from the repository.

Committing a file marked with `cvs rm` does not destroy the file's history. It simply adds a new revision, which is marked as "non-existent." The repository still has records of the file's prior contents, and can recall them as needed — for example, by `cvs diff` or `cvs log`.

There are several strategies for renaming files; the simplest is to simply rename the file in your working directory, and run `cvs rm` on the old name, and `cvs add` on the new name. The disadvantage of this approach is that the log entries for the old file's content do not carry over to the new file. Other strategies avoid this quirk, but have other, stranger problems.

You can add directories just as you would ordinary files;

2.8 Writing good log entries

If one can use `cvs diff` to retrieve the actual text of a change, why should one bother writing a log entry? Obviously, log entries can be shorter than a patch, and allow the reader to get a general understanding of the change without delving into its details.

However, a good log entry describes the *reason* the developer made the change. For example, a bad log entry for revision 1.7 shown above might say, "Convert t to lower-case." This would be accurate, but completely useless; `cvs diff` provides all the same information, more clearly. A better log entry would be, "Make this test case-insensitive," because it makes the purpose clear to anyone with a general understanding of the code: HTTP clients should ignore case differences when parsing reply headers.

2.9 Handling conflicts

As mentioned above, the `cvs update` command incorporates changes made by other developers into your working directory. If both you and another developer have modified the same file, CVS merges their changes with yours.

It's straightforward to imagine how this works when the changes apply to distant regions of a file, but what happens when you and another developer have changed the same line? CVS calls this situation a *conflict*, and leaves it up to you to resolve it.

For example, suppose that you have just added some error checking to the host name lookup code. Before you commit your change, you must run `cvs update`, to bring your sources into sync:

```
$ cvs update
cvs update: Updating .
RCS file: /u/src/master/httpc/httpc.c,v
retrieving revision 1.8
retrieving revision 1.9
Merging differences between 1.8 and 1.9 into httpc.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in httpc.c
C httpc.c
$
```

In this case, another developer has changed the same region of the file you have, so CVS complains about a conflict. Instead of printing '`M httpc.c`', as it usually does, it prints '`C httpc.c`', to indicate that a conflict has occurred in that file.

To resolve the conflict, bring up the file in your editor. CVS marks the conflicting text this way:

```
  /* Look up the IP address of the host.  */
  host_info = gethostbyname (hostname);
<<<<<<< httpc.c
  if (! host_info)
    {
      fprintf (stderr, "%s: host not found: %s\n", progname, hostname);
      exit (1);
    }
=======
  if (! host_info)
    {
      printf ("httpc: no host");
      exit (1);
    }
>>>>>>> 1.9

  sock = socket (PF_INET, SOCK_STREAM, 0);
```

The text from your working file appears at the top, after the '`<<<`' characters; below it is the conflicting text from the other developer. The revision number '`1.9`' indicates that the conflicting change was introduced in version 1.9 of the file, making it easier for you to check the logs, or examine the entire change with `cvs diff`.

Once you've decided how the conflict should be resolved, remove the markers from the code, and put it in its proper state. In this case, since your error handling code is clearly superior, you could simply throw out the other developer's change, leaving the text like this:

```
  /* Look up the IP address of the host.  */
  host_info = gethostbyname (hostname);
  if (! host_info)
    {
      fprintf (stderr, "%s: host not found: %s\n", progname, hostname);
      exit (1);
    }

  sock = socket (PF_INET, SOCK_STREAM, 0);
```

Once this is done, you can test your change and commit the file:

```
$ make
gcc -g -Wall -Wmissing-prototypes  -lnsl -lsocket  httpc.c   -o httpc
$ httpc GET http://www.cyclic.com
HTTP/1.0 200 Document follows
Date: Thu, 31 Oct 1996 23:04:06 GMT
...
```

```
$ httpc GET http://www.frobnitz.com
httpc: host not found: www.frobnitz.com
$ cvs commit httpc.c
```

It's important to understand what CVS does and doesn't consider a conflict. CVS does not understand the semantics of your program; it simply treats its source code as a tree of text files. If one developer adds a new argument to a function and fixes its callers, while another developer simultaneously adds a new call to that function, and does not pass the new argument, that is certainly a conflict — the two changes are incompatible — but CVS will not report it. CVS's understanding of conflicts is strictly textual.

In practice, fortunately, conflicts are rare. Usually, they seem to result from two developers attempting to address the same problem, a lack of communication between developers, or disagreement about the design of the program. Allocating tasks to developers in a reasonable way reduces the likelihood of conflicts.

Many version control systems allow a developer to *lock* a file, preventing others from making changes to it until she has committed her changes. While locking is appropriate in some situations, it is not clearly a better solution than the approach CVS takes. Changes can usually be merged correctly, and developers occasionally forget to release locks; in both cases, explicit locking causes unnecessary delays. Furthermore, locks prevent only textual conflicts; they do not prevent semantic conflicts of the sort described above, if the two developers make their changes to different files.