

Chapter 5

Tips and Troubleshooting

What to Do When Things Go Wrong

CVS is not "black box" software. Black boxes don't let you peek inside—you know what they do, but now how they do it. The assumption is that a black box usually doesn't need to be fixed. Most of the time, the software should work perfectly, so users don't need internal access. However, when black boxes do fail, they tend to fail completely.

CVS is instead a perfectly transparent box. Its moving parts are exposed directly to the environment, accessible to everyone, and bits of that environment (unexpected file permissions, interrupted commands, competing processes, and so on) can sometimes get inside the mechanism and gum up the gears. However, even though CVS does not always work perfectly, it rarely fails completely, either. It has the advantage of graceful degradation; the degree to which it doesn't work is usually proportional to the number and severity of problems in its environment. If you know enough about what CVS is trying to do—and how it's trying to do it—you'll know where to start looking when things go wrong.

Although we can't list all of the problems that you might encounter (there are several hundreds of possible problem constellations), we have included some of the more common ones here. This chapter is divided into two sections: The first describes those parts of the environment to which CVS is most sensitive (mainly repository permissions and the working copy administrative area), and the second describes some of the most

frequently encountered problems and their solutions. By seeing how to handle these common situations, you get a feeling for how to approach any unexpected problems in CVS.

The Usual Suspects

As a CVS administrator, you will find that most of your users' problems are caused by inconsistent working copies or incorrect repository permissions. Therefore, before looking at any specific situations, here is a quick overview of the working copy administrative area and a review of a few important issues surrounding repository permissions.

The Working Copy Administrative Area

You've already seen the basics of working copy structure in Chapter 2; in this section, we go into a bit more detail. Most of the details concern the files in the CVS/ administrative subdirectories. You already know about Entries, Root, and Repository; however, the CVS/ subdirectory can also contain other files, depending on the circumstances. We describe those other files here, partly so they don't surprise you when you encounter them and partly so you can fix them if they ever cause trouble.

CVS/Entries.Log

Sometimes, a file named "CVS/Entries.Log" will mysteriously appear. The sole purpose of this file is to temporarily cache minor changes to CVS/Entries until some operation comes along that is significant enough to be worth rewriting the entire Entries file. CVS cannot edit the Entries file in place; it must read the entire file in and write it back out to make any change. To avoid this effort, CVS sometimes records small changes in Entries.Log until the next time it needs to rewrite Entries.

The format of Entries.Log is like that of Entries, except for an extra letter at the beginning of each line. "A" means that the line is to be added to the main Entries file and "R" means it is to be removed.

For the most part, you can ignore Entries.Log; it's rare that a human has to understand the information it contains. However, if you're reading over an Entries file to debug some problem in a working copy, you should also examine Entries.Log.

CVS/Entries.Backup

The CVS/Entries.Backup file is where CVS actually writes out a new Entries file, before renaming it to "Entries" (similar to the way it writes to temporary RCS files in the repository and then renames them when they're complete). Because this file becomes Entries when it's complete, you'll rarely see an Entries.Backup file; if you do see one, it means something interrupted CVS in the middle of an operation.

CVS/Entries.Static

If the CVS/Entries.Static file exists, it means that the entire directory has not been fetched from the repository. (When CVS knows a working directory is in an incomplete state, it will not bring additional files into that directory.)

The Entries.Static file is present during checkouts and updates and is removed immediately after the operation is complete. If you see Entries.Static, it means that CVS was interrupted, and its presence prevents CVS from creating any new files in the working copy. (Often, running **cvs update -d** solves the problem and removes Entries.Static.)

Note

The absence of Entries.Static does not necessarily imply that the working copy contains all of the project's files. Whenever a new directory is created in the project's repository and someone updates their working copy without passing the ***-d*** *flag to* ***update****, the new directory will not be created in the working copy. Locally, CVS is unaware that there is a new directory in the repository, so it goes ahead and removes the Entries.Static file when the update is complete, even though the new directory is not present in the working copy.*

CVS/Tag

If the CVS/Tag file is present, it names a tag associated, in some sense, with the directory. We say "in some sense" because, as you know, CVS does not actually keep any revision history for directories and, strictly speaking, cannot attach tags to them. Tags are attached to regular files only or, more accurately, to particular revisions in regular files.

However, if every file in a directory is on a particular tag, CVS likes to think of the entire directory as being on the tag, too. For example, if you were to check out a working copy on a particular branch:

```
yarkon$ cvs co -r Bugfix_Branch_1
```

and then add a file inside it, you'd want the new file's initial revision to be on that branch, too. For similar reasons, CVS also needs to know if the directory has a nonbranch sticky tag or date set on it.

Tag files contain one line. The first character on the line is a single-letter code telling what kind of tag it is, and the rest of the line is the tag's name. Currently, CVS uses only these three single-letter codes:

- **T**—A branch tag
- **N**—A nonbranch (regular) tag

- **D**—A sticky date, which occurs if a command such as

```
yarkon$ cvs checkout -D 2001-05-15 myproj
```

or

```
yarkon$ cvs update -D 2001-05-15 myproj
```

is run.

(If you see some other single-letter code, it just means that CVS added a new tag type after this chapter was written.)

You should not remove the Tag file manually; instead, use **cvs update -A**.

Rarities

There are a few other files you might occasionally find in a CVS/ subdirectory:

- CVS/Checkin.prog, CVS/Update.prog
- CVS/Notify, CVS/Notify.tmp
- CVS/Base/, CVS/Baserev, CVS/Baserev.tmp
- CVS/Template

These files are not usually the cause of problems; we list them here just for your information.

Portability and Future Extension

As features are added to CVS, new files (not listed here) might appear in working copy administrative areas. As new files are added, they'll probably be documented in the Cederqvist manual, in the node "Working Directory Storage." You can also start looking in src/cvs.h in the source distribution, if you prefer to learn from code.

Finally, note that all CVS/* files—present and future—use whatever line-ending convention is appropriate for the working copy's local system (for example, "LF" for Unix or "CRLF" for Windows). This means that if you transport a working copy from one kind of machine to another, CVS won't be able to handle it (but then you'd have other problems, because the revision-controlled files themselves would have the wrong line-end conventions for their new location).

Repository Permissions

CVS does not require any particular repository permission scheme; it can handle a wide variety of permission arrangements. However, to avoid getting confusing behaviors, you should make sure your repository setup meets at least the following criteria:

- If a user wants any kind of access at all to a given subdirectory of the repository —even read-only access—she usually needs file system-level write permission to that subdirectory.

This level of permission is necessary because CVS creates temporary lock files in the repository to ensure data consistency. Even read-only operations (such as **checkout** or **update**) create locks, to signal that they need the data to stay in one state until they're done.

As noted previously, you can get around this writability requirement by setting the **LockDir** parameter in CVSROOT/config, like this:

```
LockDir=/usr/local/cvslocks
```

Of course, then you would need to make sure the directory /usr/local/cvslocks is writable by all CVS users. Either way, most CVS operations, including read-only ones, are going to require a writable directory somewhere. By default, that directory is the project's repository; if you're very security conscious, you can change it to somewhere else.

- Make sure the CVSROOT/history file is world-writable (if it exists at all). If the history file exists, most CVS operations attempt to append a record to it; if the attempt fails, the operation exits with an error.

  Unfortunately (and inexplicably), the history file is not born world-writable when you create a new repository with **cvs init**. At least with the current version of CVS, you should explicitly change the history file permissions after you create a new repository (or just remove the file, if you want to disable history logging entirely).

- For security purposes, you almost certainly want to make sure that most CVS users do not have Unix-level write access to the CVSROOT directory in the repository. If someone has checkin access to CVSROOT, they can edit commitinfo, loginfo, or any of the other trigger files to invoke a program of their choice—they could even commit a new program if the one they want isn't already on the system. Therefore, you should assume that anyone who has commit access to CVSROOT is able to run arbitrary commands on the system.

Common Problems and How to Solve Them

The rest of this chapter is organized into a series of questions and answers, similar to an Internet FAQ (Frequently Asked Questions) document. All the questions and answers here are based on actual CVS experiences. But before we look at specific cases, let's take a moment to consider CVS troubleshooting from a general point of view.

The first step in solving a CVS problem is usually to determine whether it's a working copy or a repository problem. The best technique for doing that, not surprisingly, is to see if the problem occurs in working copies other than the one where it was first noticed. If it does, it's likely a repository issue; otherwise, it's probably just a local issue.

Working copy problems tend to be encountered more frequently, not because working copies are somehow less reliable than repositories, but because each repository usually has many

working copies. Although most working copy knots can be untied with enough patience, you might occasionally find it more time-efficient simply to delete the working copy and check it out again.

Of course, if checking it out again takes too long, or there is considerable information in an uncommitted state in the working copy that you don't want to lose, or if you just want to know what's wrong, it's worth digging around to find the cause of the problem. When you start digging around, one of the first places to look is in the CVS/ subdirectories. Check the file contents and the file permissions. Very occasionally, the permissions can mysteriously become read-only or even unreadable. (We suspect this is caused by users accidentally mistyping Unix commands rather than any mistake on the part of CVS.)

Repository problems are almost always caused by incorrect file and directory permissions. If you suspect that a problem might be due to bad repository permissions, first find out the effective repository user ID of the person who's having the trouble. For all local and most remote users, this is either their regular username or the username they specified when they checked out their working copy. If they're using the **pserver** method with user-aliasing (see the section "The Password Authenticating Server" in Chapter 3), the effective user ID is the one on the right in the CVSROOT/passwd file. Failure to discover this early on can cause you to waste a lot of time debugging the wrong thing.

And now, without further ado, here are some real-life troubleshooting examples.

Some Real-Life Problems, with Solutions

All of the situations mentioned in this section are ones we have encountered in our real-life adventures as CVS troubleshooters. (A couple of items are not really problems, but are questions that we have heard so often that they might as well be answered here.) The list is meant to be fairly comprehensive, so it might repeat material you've seen in earlier chapters.

The situations are listed according to how frequently they seem to arise, with the most common ones first.

I Keep Getting Messages about Waiting for Locks. What's Going On?

If you see a message like this

```
cvs update: [22:58:26] waiting for qsmith's lock in /usr/local/newrepos/myproj
```

it means you're trying to access a subdirectory of the repository that is locked by some other CVS process at the moment. Because a process is running in that directory, it might not be in a consistent state for other CVS processes to use.

However, if this message persists for a long time, it probably means that a CVS process failed to clean up after itself, for whatever reason. It can happen when CVS dies suddenly and unexpectedly—for example, due to a power failure on the repository machine.

The solution is to remove the lock files by hand from the repository subdirectory in question. Go into that part of the repository and look for files named "#cvs.lock" or that begin with "#cvs.wfl" or "#cvs.rfl". Compare the file's timestamps with the start times of any currently running CVS processes. If the files could not possibly have been created by any of those processes, it's safe to delete them. The waiting CVS processes eventually notice when the lock files are gone—this should take about 15 seconds—and allow the requested operation to proceed.

See the node "Locks" in the Cederqvist manual for more details.

CVS Claims a File Is Failing Up-to-Date Check. What Should I Do?

Don't panic—it just means that the file has changed in the repository since the last time you checked it out or updated it.

Run **cvs update** on the file to merge in the changes from the repository. If the received changes conflict with your local changes, edit the file to resolve the conflict. Then try your commit again—it will succeed, barring the possibility that someone committed yet another revision while you were busy merging the last changes.

*I Can't Seem to Get the **pserver** Access Method to Work.*

The most common, least obvious cause of this problem is that you forgot to list the repository using an **– –allow-root** option in your inetd configuration file.

Recall this example /etc/inetd.conf line from Chapter 3:

```
cvspserver stream tcp nowait root /usr/local/bin/cvs cvs \
          --allow-root=/usr/local/newrepos pserver
```

(In the actual file, this is all one long line, with no backslash.)

The **– –allow-root=/usr/local/newrepos** portion is a security measure, to make sure that people can't use CVS to get **pserver** access to repositories that are not supposed to be served remotely. Any repository intended to be accessible via **pserver** must be mentioned in an **– –allow-root**. You can have as many different **– –allow-root** options as you need for all of your system's repositories (or as many as you want until you bump up against your inetd's argument limit).

*The **pserver** Access Method STILL Isn't Working!*

Okay, if the problem is not a missing **– –allow-root**, here are a few other possibilities:

- The user has no entry in the CVSROOT/passwd file, and the CVSROOT/config file has **SystemAuth=no** so CVS will not fall back on the system password file (or **SystemAuth=yes**, but the system password file has no entry for this user, either).
- The user has an entry in the CVSROOT/passwd file, but there is no user by that name on the system, and the CVSROOT/passwd entry does not map the user to any valid system username.

- The password is wrong (CVS is usually pretty good about informing the user of this, so that's probably not the answer).
- Everything is set up correctly with the passwd files and in /etc/inetd.conf, but you forgot an entry like this in /etc/services:

```
cvspserver        2401/tcp
```

  so inetd is not even listening on that port to pass connections off to CVS.

My Commits Seem to Happen in Pieces, Not Atomically.

That's because CVS, unlike modern databases, does not have the notion of atomical transactions. (All modern database management systems allow for atomical transactions; changes are either committed for good or rolled back to the original state. A record can never be in a state in between.) More specifically, CVS operations happen directory by directory. When you do a commit (or an update, or anything else, for that matter) spanning multiple directories, CVS locks each corresponding repository directory in turn while it performs the operation for that directory.

For small- to medium-sized projects, this is rarely a problem—CVS manages to do its thing in each directory so quickly that you never notice the lack of atomicity. Unfortunately, in large projects, scenarios like the following can occur (imagine this taking place in a project with at least two deep, many-filed subdirectories, A and B):

1. User qsmith runs **commit**, involving files from both subdirectories. CVS commits the files in B first (perhaps because qsmith specified the directories on the command line in that order).
2. User jrandom runs **cvs update**. The update, for whatever reason, starts with working copy directory A. (CVS makes no guarantees about the order in which it processes directories or files, if left to its own devices.) Note that there is no locking contention, because qsmith is not active in A yet.
3. User qsmith's commit finishes B, moves on to A, and finishes A.
4. Finally, jrandom's update moves on to B and finishes it.

Clearly, when this is all over, jrandom's working copy reflects qsmith's changes to B but not A. Even though qsmith intended the changes to be committed as a single unit, it didn't happen that way. Now jrandom's working copy is in a state that qsmith never anticipated.

The solution, of course, is for jrandom to do another **cvs update** to fetch the uncaught changes from qsmith's commit. However, that assumes that jrandom has some way of finding out in the first place that she got only part of qsmith's changes.

There's no easy solution to this quandary. You simply have to hope that the inconsistent state of the working copy will somehow become apparent (maybe the software won't build,

or jrandom and qsmith will have a conversation that's confusing until they realize what must have happened).

The failure of CVS to provide atomic transaction guarantees is certainly a bug. The only reason that locks are not made at the top level of the repository is that this would result in intolerably frequent lock contentions for large projects with many developers. Therefore, CVS has chosen the lesser of two evils: reducing the contention frequency while allowing the possibility of interleaved reads and writes. Someday, someone might modify CVS (say, by speeding up repository operations) so that it doesn't have to choose between two evils; until then, we're stuck with nonatomic actions.

For more information, see the node "Concurrency" in the Cederqvist manual.

CVS Keeps Changing the Permissions of My Files. Why Does It Do That?

In general, CVS is not very good at preserving permissions on files. When you import a project and then check it out, there is no guarantee that the file permissions in the new working copy will be the same as when the project was imported. More likely, the working copy files will be created with the same standard permissions that you normally get on newly created files.

However, there is at least one exception. If you want to store executable shell scripts in the project, you can keep them executable in all working copies by making the corresponding repository file executable:

```
yarkon$ ls -l /usr/local/newrepos/someproj
total 6
-r--r--r--   1 jrandom  users         630 Aug 17 01:10 README.txt,v
-r-xr-xr-x   1 jrandom  users        1041 Aug 17 01:10 scrub.pl,v*
-r--r--r--   1 jrandom  users         750 Aug 17 01:10 hello.c,v
```

Notice that although the file is executable, it is still read-only, as all repository files should be. (Remember that CVS works by making a temporary copy of the RCS file, doing everything in the copy, and then replacing the original with the copy when ready.)

When you import or add an executable file, CVS preserves the executable bits, so if the permissions were correct from the start, you have nothing to worry about. However, if you accidentally add the file before making it executable, you must go into the repository and manually set the RCS file to be executable.

Note

The repository permissions always dominate. If the file is nonexecutable in the repository, but executable in the working copy, the working copy file will also be nonexecutable after you do an update. Having your files' permissions silently change can be extremely frustrating. If this happens, first check the repository and see if you can solve it by setting the appropriate permissions on the corresponding RCS files.

A "PreservePermissions" feature that might alleviate some of these problems has recently been added to CVS. However, using this feature can cause other unexpected results (which is why we are not recommending it unconditionally). Make sure you read the nodes "config" and "Special Files" in the Cederqvist before putting **PreservePermissions=yes** in CVSROOT/config.

CVS on Windows Complains It Can't Find My .cvspass File. Why?

For **pserver** connections, CVS on the client side tries to find the .cvspass file in your home directory. Windows machines don't have a natural "home" directory, so CVS consults the artificial environment variable %HOME%. However, you have to be very careful about how you set **HOME**. The following works:

```
set HOME=C:
```

This does not work:

```
set HOME=C:\
```

That final backslash is enough to confuse CVS, and it will be unable to open C:\.cvspass.

So, the quick and permanent solution is to put the correct variable (HOME=C:) into your autoexec.bat and reboot. CVS **pserver** should work fine after that.

My Working Copy Is on Several Different Branches. Help!

If different subdirectories of your working copy somehow got on different branches, you probably ran updates with the **-r** flag, but from places other than the top level of the working copy.

No big deal. If you want to return to the trunk, just run this

```
cvs update -r HEAD
```

or this:

```
cvs update -A
```

from the top directory. If you want to put the whole working copy on one of the branches, do this:

```
cvs update -r Branch_Name
```

There's nothing wrong with having one or two subdirectories of your working copy on a different branch than the rest, if you need to do some temporary work on that branch just in those locations. However, it's usually a good idea to switch them back when you're finished—life is much less confusing when your whole working copy is on the same line of development.

*When I Do an **export -D**, It Sometimes Seems to Miss Recent Commits!*

This is due to a clock difference between the repository and local machines. You can solve it by resetting one or both of the clocks, or by specifying a different date as the argument to **-D**. It's perfectly acceptable to specify a date in the future (such as **-D tomorrow**), if that's what it takes to compensate for the time difference.

I'm Having Problems with Sticky Tags; I Just Want to Get Rid of Them.

Various CVS operations cause the working copy to have a *sticky tag*, which is a single tag that corresponds to each revision for each file. (In the case of a branch, the sticky tag is applied to any new files added in the working copy.) You get a sticky tagged working area whenever you check out or update by tag or date, for example

```
yarkon$ cvs update -r Tag_Name
```

or:

```
yarkon$ cvs checkout -D "2001-08-16"
```

If a date or a nonbranch tag name is used, the working copy will be a frozen snapshot of that moment in the project's history, so (naturally) you will not be able to commit any changes from it.

To remove a sticky tag, run **update** with the **-A** flag

```
yarkon$ cvs update -A
```

which clears all the sticky tags and updates each file to its most recent trunk revision.

CVS Checkout/Update Exits with Error, Saying It Cannot Expand Modules.

This is just a case of a bad error message in CVS; probably someone will get around to fixing it sooner or later, but meanwhile it can bite you.

The error message looks something like this:

```
yarkon$ cvs co -d bwf-misc user-space/bwf/writings/misc
cvs server: cannot find module 'user-space/bwf/writings/misc' - ignored
cvs [checkout aborted]: cannot expand modules
```

CVS appears to be saying that there's something wrong with the CVSROOT/modules file. However, what's really going on is a permission problem in the repository. The directory you are trying to check out isn't readable, or one of its parents isn't readable. In this case, it was a parent that wasn't readable:

```
yarkon$ ls -ld /usr/local/cvs/user-space/bwf

drwx------  19 bwf      users        1024 Aug 17 01:24 bwf/
```

Don't let that egregiously wrong error message fool you—this is a repository permission problem.

I Can't Seem to Turn Off Watches!

You probably did

```
yarkon$ cvs watch remove
```

on all the files, but forgot to also do:

```
yarkon$ cvs watch off
```

A hint for diagnosing watch problems: Sometimes it can be immensely clarifying just to go into the repository and examine the CVS/fileattr files directly.

My Binary Files Are Messed Up.

Did you remember to use **-kb** when you added them? If not, CVS might have performed line-end conversion or RCS keyword substitution on them. The easiest solution is usually to mark them as binary

```
yarkon$ cvs admin -kb foo.gif
```

and then commit a fixed version of the file. CVS will not corrupt the new commit or any of the commits thereafter, because it now knows the file is binary.

CVS Isn't Doing Line-End Conversion Correctly.

If you're running the CVS client on a non-Unix platform and are not getting the line-end conventions that you want in some working copy files, it's usually because the files were accidentally added with **-kb** when they shouldn't have been. This can be fixed in the repository with, believe it or not, the command:

```
yarkon$ cvs admin -kkv FILE
```

The **-kkv** option causes normal keyword substitution and implies normal line-end conversions as well. (Internally, CVS is a bit confused about the difference between keyword substitution and line-end conversion. This confusion is reflected in the way the **-k** options can control both parameters.)

Unfortunately, that **admin** command fixes only the file in the repository—your working copy still thinks the file is binary. You can hand edit the **CVS/Entries** line for that file, removing the **-kb**, but that won't solve the problem for any other working copies out there.

How Do I Remove a Subdirectory in My Project?

Well, you can't exactly remove the subdirectory, but you can remove all of the files in it (first remove them, then **cvs remove** them, and then run **commit**). Once the directory is

empty, you can have it automatically pruned out of your working copies by passing the **-P** flag to **update**.

Can I Copy .cvspass Files or Portions of Them?

Yes, you can. You can copy .cvspass files from machine to machine, and you can even copy individual lines from one .cvspass file to another. For high-latency servers, this might be faster than running **cvs login** from each working copy machine.

Remember that if you transport a .cvspass file between two machines with different line-ending conventions, it probably won't work. (Of course, you can probably do the line-end conversion manually without too much trouble.)

I Just Committed Some Files with the Wrong Log Message.

You don't need to hand-edit anything in the repository to solve this—just run **admin** with the **-m** flag. Remember to have no space between **-m** and its argument, and to quote the replacement log message as you would a normal one:

```
yarkon$ cvs admin -m1.17:"I take back what I said about the customer." hello.c
```

I Need to Move Files around without Losing Revision History.

In the repository, copy (don't move) the RCS files to the desired new location in the project. They must remain in their old locations as well.

Then, in a working copy, do:

```
yarkon$ rm oldfile1 oldfile2 ...
yarkon$ cvs remove oldfile1 oldfile2 ...
yarkon$ cvs commit -m "removed from here" oldfile1 oldfile2 ...
```

When people do updates after that, CVS correctly removes the old files and brings the new files into the working copies just as though they had been added to the repository in the usual way (except that they'll be at unusually high revision numbers for supposedly new files).

How Can I Get a List of All Tags in a Project?

Currently, there is no convenient way to do this in CVS—a lack that is sorely felt by all users. We believe work is under way to make this feature available. By the time you read this, a **cvs tags** command or something similar might be available.

Until then, there are workarounds. You can run **cvs log -h** and read the sections of the output following the header **symbolic names:**. Or, if you happen to be on the repository machine, you can just look directly in the repository at the beginnings of some of the RCS files. All of the tags (branches and nonbranches) are listed in the **symbols** field:

```
yarkon$ head /usr/local/newrepos/hello.c,v
head        2.0;
```

```
access;
symbols
    Release_1_0:1.22
    Exotic_Greetings-2:1.21
    merged-Exotic_Greetings-1:1.21
    Exotic_Greetings-1:1.21
    merged-Exotic_Greetings:1.21
    Exotic_Greetings-branch:1.21.0.2
    Root-of-Exotic_Greetings:1.21
    start:1.1.1.1
    jrandom:1.1.1;
locks; strict;
comment    @ * @;
```

How Can I Get a List of All Projects in a Repository?

As with getting a list of tags, this feature is not implemented in the most current version of CVS, but it's highly likely that it will be implemented soon. We believe the command will be called **cvs list** with a short form of **cvs ls**, and it will probably both parse the modules file and list the repository subdirectories.

In the meantime, examining the CVSROOT/modules file (either directly or by running **cvs checkout -c**) is probably your best bet. However, if no one has explicitly made a module for a particular project, it won't show up.

Some Commands Fail Remotely but not Locally. How Should I Debug?

Sometimes there's a problem in the communication between the client and the server. If so, it's a bug in CVS, but how would you go about tracking down such a thing?

CVS gives you a way to watch the protocol between the client and server. Before you run the command on the local (working copy) machine, set the environment variable **CVS_CLIENT_LOG**. Here's how to do this in Bourne shell syntax:

```
yarkon$ CVS_CLIENT_LOG=clog; export CVS_CLIENT_LOG
```

Once that variable is set, CVS will record all communications between client and server in two files whose names are based on the variable's value:

```
yarkon$ ls
CVS/        README.txt    a-subdir/    b-subdir/    foo.gif      hello.c
yarkon$ cvs update
? clog.in
? clog.out
cvs server: Updating .
cvs server: Updating a-subdir
cvs server: Updating a-subdir/subsubdir
```

```
cvs server: Updating b-subdir
yarkon$ ls
CVS/              a-subdir/    clog.in     foo.gif
README.txt        b-subdir/    clog.out    hello.c
yarkon$
```

The clog.in file contains everything that the client sent into the server; clog.out, on the other hand, contains everything the server sent back out to the client. Here are the contents of clog.out, to give you a sense of what the protocol looks like:

```
Valid-requests Root Valid-responses valid-requests Repository           \
Directory Max-dotdot Static-directory Sticky Checkin-prog Update-prog   \
Entry Kopt Checkin-time Modified Is-modified UseUnchanged Unchanged     \
Notify Questionable Case Argument Argumentx Global_option Gzip-stream   \
wrapper-sendme-rcsOptions Set expand-modules ci co update diff log add  \
remove update-patches gzip-file-contents status rdiff tag rtag import   \
admin export history release watch-on watch-off watch-add watch-remove  \
watchers editors init annotate noop
ok
M ? clog.in
M ? clog.out
E cvs server: Updating .
E cvs server: Updating a-subdir
E cvs server: Updating a-subdir/subsubdir
E cvs server: Updating b-subdir
ok
```

The clog.in file is even more complex, because it has to send revision numbers and other per-file information to the server.

There isn't space here to document the client/server protocol, but you can read the "cvsclient" Info pages that are distributed with CVS for a complete description. You might be able to figure out a good deal of it just from reading the raw protocol itself. Although you probably won't find yourself using client logging until you've eliminated all of the other possible causes of a problem, it is an invaluable tool for finding out what's really going on between the client and server.

I Don't See My Problem Covered in this Chapter.

Email an accurate and complete description of your problem to **info-cvs@gnu.org**, the CVS discussion list. Its members are located in many different time zones, and we've usually gotten a response within an hour or two of sending a question. Please join the list by sending email to **info-cvs-request@gnu.org**, so you can help answer questions, too.

I Think I've Discovered a Bug in CVS. What Should I Do?

CVS is far from perfect—if you've already tried reading the manual and posting a question on the mailing list, and you still think you're looking at a bug, you probably are.

Send as complete a description of the bug as you can to **bug-cvs@gnu.org**. (You can also subscribe to that list; just use **bug-cvs-request** instead.) Be sure to include the CVS version number (both client and server versions, if applicable), and a recipe for reproducing the bug.

If you have written a patch to fix the bug, include it and mention on the subject line of your message that you have a patch. The maintainers will be very grateful.

(Further details about these procedures are outlined in the node "Bugs" in the Cederqvist manual and the file HACKING in the source distribution.)

I've Implemented a New Feature in CVS. To Whom Do I Send It?

Same as with a bug: Send the patch to **bug-cvs@gnu.org**. Make sure you've read over the HACKING file first, though.

Things Change

The troubleshooting techniques and known bugs described in this chapter are accurate as of (approximately) CVS version 1.11.5. Things move fast in the CVS world, however.

SourceGear is always busy organizing and cleaning up various patches that have been floating around, with the intention of incorporating many of them into CVS. Some of these patches will probably fix bugs listed in this chapter, and others might provide new troubleshooting tools for CVS users.

The best way to stay up to date with what's going on is to read the NEWS file in your CVS distribution, watch the mailing lists, and look for changes to the Cederqvist manual and the online version of some of the chapters of this book (see the last section of this book's Introduction for specific chapter numbers).