

Chapter 3

CVS Repository Administration

The Administrator's Role

In the last chapter, you learned enough CVS to use it effectively as a project participant. There is, however, quite a difference between being a project manager and being a participant. If you're going to be a project maintainer, you need to know how to install CVS and administer repositories.

In this chapter, we look in detail at how the repository is structured and how CVS uses it. You'll learn all the major steps CVS goes through during updates and commits, and how you can modify its behavior. By understanding *how* CVS works, you'll also be able to trace problems to their causes and fix them in maintainable ways.

This sounds complicated, and to some extent it really is. However, remember that CVS has been serving the developer community for quite some time and will probably be around for many years to come. That's the beauty of CVS: whatever you learn now will be useful for a long time. CVS also tends to become more indispensable the more you use it. If you're going to be that dependent on something (and trust us, you will be), it's worth really getting to know it.

Let's just jump right into it and see how to install CVS on your system.

Getting and Installing CVS

With the pervasiveness of Linux distributions at the time of this writing, you might find that you actually don't have to install CVS. In fact, all current Linux, Mac OSX, and BSD distributions include a

functioning and configured CVS; it's probably already installed in /usr/bin or some other likely location. If not, Red Hat, Mandrake, and other compatible Linux users can usually find an RPM (Red Hat Package Manager) for the latest (or nearly latest) version of CVS in their distributions or from **www.rpmfind.net**. Debian users, on the other hand, can install the latest Debian package with these commands:

```
yarkon$ apt-get update
yarkon$ apt-get install cvs
```

Building CVS from Source

If CVS isn't already on your machine, you'll probably have to build it from source. If you're a non-Unix user, you'll probably find it easier to get a prebuilt binary for your operating system (more on that later in this chapter). Fortunately, CVS is fully *autoconfigured*—that is, it uses the GNU autoconfiguration mechanism, making compilation from source surprisingly easy.

As of this writing, there are two canonical sites from which you can download CVS. One is the Free Software Foundation's FTP site, **ftp://ftp.gnu.org/gnu/**, which offers CVS as an official GNU tool. The other is the official CVS site. It distributes releases from **ftp://ftp.cvshome.org/pub/**.

Either location is fine. In the following example, we use the official CVS site. If you point your FTP client (probably your Web browser) to **http://ftp.cvshome.org//**, you'll see a list of directories, something like this:

```
LATEST/                         13-Jun-2001 07:56      -
LATEST_IS_1.11.1p1              26-Apr-2001 08:51     0k
README                          13-Jun-2001 07:46     1k
cvs-1.10.5/                     26-Apr-2001 18:55      -
cvs-1.10.6/                     04-Apr-2001 08:40      -
cvs-1.10.7/                     04-Apr-2001 08:41      -
cvs-1.10.8/                     26-Apr-2001 18:59      -
cvs-1.10/                       26-Apr-2001 18:49      -
cvs-1.11.1/                     13-Jun-2001 07:56      -
cvs-1.11/                       26-Apr-2001 19:00      -
cvs-1.9.28/                     26-Apr-2001 18:59      -
cvs-1.9/                        26-Apr-2001 11:28      -
linux/                          27-Apr-2001 14:10      -
macintosh/                      04-Apr-2001 08:41      -
os2/                            26-Apr-2001 18:59      -
rcs/                            26-Apr-2001 19:01      -
tkcvs/                          04-Apr-2001 08:42      -
training/                       04-Apr-2001 08:42      -
unix/                           28-Apr-2001 12:44      -
```

```
vms/                            04-Apr-2001 08:42       -
win32/                          26-Apr-2001 19:00       -
```

Pay attention to the directories beginning with "cvs-" (you can ignore most of the others). As you can see, there are nine cvs- directories, which means that you're already faced with a choice: Get the designated "stable" release, or go with a newer (but less-tested) interim release. The stable releases have only one decimal point, as in "cvs-1.11," whereas the interim releases have minor version increments tacked on the end, as in "1.10.7."

Note

The GNU site usually offers only the major releases, not the interim ones, so you won't see all of this if you get CVS from there.

In general, the interim releases are pretty safe, and they sometimes contain fixes to bugs that were found in the major release. Your best policy is to go with the highest interim release; however, if you encounter any problems with it, be prepared to drop back to the previous release.

The highest release listed in the earlier example is cvs-1.11. Entering that directory, we see this:

```
Index of /pub/cvs-1.11
    cvs-1.11.tar.gz      17-May-01 03:41   2.4M
```

That's it—the full source code to CVS. Just download it to your machine, and you're ready to build. At this point, if you're already familiar with the standard build process for GNU tools, you know what to do and probably don't need to read anything between here and the section "Anatomy of a CVS Distribution." On the other hand, if you're not sure how to proceed, read on.

Compilation Instructions

The following compilation instructions and examples assume that you have a fairly standard distribution of Unix. Any of the free versions of Unix (for example, FreeBSD or Linux) should work with no problem, as should the major commercial Unix versions (such as SunOS/Solaris, AIX, HP-UX, or True64). Even if these instructions don't work for you exactly as written, don't give up hope. Although covering the details of compiling on every operating system is beyond the scope of this book, later in this chapter we give you some pointers to other help resources.

To proceed with the compilation, first unpack the tar file using GNU gunzip and tar (if you don't have these installed on your system, you can get gunzip from **ftp://ftp.cvshome.org**):

```
yarkon$ gunzip cvs-1.11.tar.gz
yarkon$ tar xvf cvs-1.11.tar
```

Now you have a new directory on your machine—cvs-1.11—and it is populated with the CVS source code. Go into that directory and configure CVS for your system by using the provided **configure** script:

```
yarkon$ cd cvs-1.11
yarkon$  ./configure
creating cache ./config.cache
checking for gcc... gcc
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking how to run the C preprocessor... gcc -E
  (etc)
```

When the **configure** command finishes, the source tree will know everything it needs to know about compiling on your machine. The next step is to type:

```
yarkon$ make
```

You'll see lots of output fly by, then type:

```
yarkon$ make install
```

You will probably need to do that last step as the superuser. You'll see yet more output fly by; when it's all over, CVS will be installed on your system.

By default, the CVS executable ends up as /usr/local/bin/cvs. This assumes you have a decent **make** program installed on your system (again, if you don't have one, get the GNU project's **make** from **ftp://ftp.gnu.org/gnu/make/**).

If you want CVS to install to a location other than /usr/local/bin, you should change how you run the initial configuration step. For example,

```
yarkon$ ./configure --prefix=/usr
```

results in CVS being installed as /usr/bin/cvs (it always ends up as *PREFIX*/bin/cvs). The default prefix is /usr/local, which is fine for most installations.

Note

For experienced users: Although older versions of CVS consisted of more than just an executable in that they depended on having RCS installed as well, this has not been the case since version 1.10. Therefore, you don't need to worry about any libraries or executables other than cvs itself.

If you intend to use CVS to access only remote repositories, the preceding is all you need to do. If you also plan to serve a repository from *this* system, a few additional steps are necessary, which are covered later in this chapter.

Getting and Installing CVS under Windows

Unless you're truly religious about having the source code to your executable, you don't need to compile CVS from source on your Windows box. Unlike on a Unix system, the necessary compilation tools probably do not already exist on your system, so a source build would involve first going out and getting those tools. Because such a project is beyond the scope of this book, we just give instructions for getting a precompiled CVS binary.

First, note that Windows binary distributions of CVS are usually made only for major releases of CVS—not for the interim releases—and you won't find them on the GNU FTP site. You'll need to go to Cyclic Software's download site. In the major version directory, **http://ftp.cvshome.org/cvs-1.11/**, you'll see an extra subdirectory,

```
Index of /pub/cvs-1.11
    cvs-1.11.tar.gz        18-Jun-00 09:35   2.4M
    windows/
```

inside of which is a ZIP file:

```
Index of /pub/cvs-1.10/windows
    cvs-1.11-win.zip       18-Jun-00 10:08   589k
```

This ZIP file contains a binary distribution of CVS. Download and extract that ZIP file:

```
yarkon$ unzip cvs-1.10-win.zip

Archive:  cvs-1.10-win.zip
  inflating: cvs.html
  inflating: cvs.exe
  inflating: README
  inflating: FAQ
  inflating: NEWS
  inflating: patch.exe
  inflating: win32gnu.dll
```

The README contains detailed instructions. For most installations, they can be summarized as follows: Put all of the EXE and DLL files in a directory in your PATH. Additionally, if you're going to be using the **pserver** method to access a remote repository, you might need to put the following in your C:\autoexec.bat file and reboot:

```
set HOME=C:
```

This tells CVS where to store the .cvspass file.

At this time, CVS running under Windows cannot serve repositories to remote machines; it can be a client (connecting to remote repositories) and operate in local mode (using a

repository on the same machine). For the most part, this book assumes that CVS under Windows is operating as a client. However, it shouldn't be too hard to set up a local repository under Windows after reading the Unix-oriented instructions in the rest of this chapter.

If you are accessing only remote repositories, you might not even need to run CVS. A tool called WinCvs implements only the client-side portion of CVS. It is distributed separately from CVS itself but, like CVS, is freely available under the GNU General Public License. More information is available from **www.wincvs.org**.

Getting and Installing CVS on a Macintosh

CVS is available for the Macintosh, but not as part of the main distribution. At this time, there are actually three separate Macintosh CVS clients available:

- *MacCvs*—**www.wincvs.org**
- *MacCVSClient*—**www.glink.net.hk/~jb/MacCVSClient** or **http://cvshome.org/dev/codemac.html**
- *MacCVS Pro*—**www.maccvs.org**

Frankly, these are all equivalent. Try them all and see which one you like. MacCVS Pro seems to be under active development. MacCvs is apparently a companion project of WinCVS and shares a home page with it.

Limitations of the Windows and Macintosh Versions

The Windows and Macintosh distributions of CVS are generally limited in functionality. All of them can act as clients, meaning that they can contact a repository server to obtain a working copy, commit, update, and so on. However, they can't serve repositories themselves. If you set it up right, the Windows port can use a local-disk repository, but it still can't serve projects from that repository to other machines. In general, if you want to have a network-accessible CVS repository, you must run the CVS server on a Unix box.

Anatomy of a CVS Distribution

The preceding instructions are designed to get you up and running quickly, but there's a lot more inside a CVS source distribution than just the code. Here's a quick roadmap to the source tree, so you'll know which parts are useful resources and which can be ignored.

Informational Files

In the top level of the distribution tree, you'll find several files containing useful information (and pointers to further information). They are, in approximate order of importance:

- *NEWS*—This file lists the changes from one release to the next, in reverse chronological order (that is, most recent first). If you've already been using CVS for a while and have

just upgraded to a new version, you should look at the NEWS file to see what new features are available. Also, although most changes to CVS preserve backward compatibility, noncompatible changes do occur from time to time. It's better to read about them here than be surprised when CVS doesn't behave the way you expect it to.

- *BUGS*—This file contains exactly what you think it does: a list of known bugs in CVS. They usually aren't showstoppers, but you should read over them whenever you install a new release.

- *DEVEL-CVS*—This file is the CVS "constitution." It describes the process by which changes are accepted into the main CVS distribution and the procedures through which a person becomes a CVS developer. You don't really need to read it if you just want to use CVS; however, it's highly interesting if you want to understand how the mostly uncoordinated efforts of people scattered across the globe coalesce into a working, usable piece of software. And, of course, it's required reading if you plan to submit a patch (whether it's a bug fix or new feature) to CVS.

- *HACKING*—Despite its name, the HACKING file doesn't say much about the design or implementation of CVS. It's mainly a guide to coding standards and other technical "administrivia" for people thinking of writing a patch to CVS. You can think of it as an addendum to the DEVEL-CVS file. After you understand the basic philosophy of CVS development, you must read the HACKING file to translate that into concrete coding practices.

- *FAQ*—This is the CVS "Frequently Asked Questions" document. Unfortunately, it has a rather spotty maintenance history. David Grubbs took care of it until 1995, then he (presumably) got too busy and it languished for a while. Eventually, in 1997, Pascal Molli took over maintenance. Molli also didn't have time to maintain it by hand, but at least he found time to put it into his automated FAQ-O-Matic system, which allows the public to maintain the FAQ in a decentralized manner (basically, anyone can edit or add entries via a Web form). This was probably a good thing, in that at least the FAQ was once again being maintained; however, its overall organization and quality control are not as good as they would be if one person were maintaining it.

  The master version of the FAQ is always available from Molli's Web site (**www.loria.fr/~molli/cvs-index.html**, under the link "Documentation"). The FAQ file shipped with CVS distributions is generated automatically from that FAQ-O-Matic database, so by the time it reaches the public it's already a little bit out of date. Nevertheless, it can be quite helpful when you're looking for hints and examples of how to do something specific (say, merging a large branch back into the trunk or resurrecting a removed file). The best way to use it is as a reference document; you can bring it up in your favorite editor and do text searches on terms that interest you. Trying to use it as a tutorial would be a mistake—it's missing too many important facts about CVS to serve as a complete guide.

Subdirectories

The CVS distribution contains a number of subdirectories. In the course of a normal installation, you won't have to navigate among them, but if you want to go poking around in the sources, it's nice to know what each one does. Here they are:

```
contrib/
diff/
doc/
emx/
lib/
man/
os2/
src/
tools/
vms/
windows-NT/
zlib/
```

You can ignore the majority of these. The emx/, os2/, vms/, and windows-NT/ subdirectories all contain operating-system–specific source code, which you will need only if you're actually trying to debug a code-level problem in CVS (an unlikely situation, although not unheard of). The diff/ and zlib/ subdirectories contain CVS's internal implementations of the **diff** program and the GNU gzip compression library, respectively. (CVS uses the latter to reduce the number of bits it has to send over the network when accessing remote repositories.)

The contrib/ and tools/ subdirectories contain free third-party software that is intended to be used with CVS. In contrib/, you will find an assortment of small, specialized shell scripts (read contrib/README to find out what they do). The tools/ subdirectory used to contain contributed software, but now contains a README file, which says in part:

```
This subdirectory formerly contained tools that can be used with CVS.
In particular, it used to contain a copy of pcl-cvs version 1.x.
Pcl-cvs is an Emacs interface to CVS.

If you are looking for pcl-cvs, we'd suggest pcl-cvs version 2.x, at:
    ftp://ftp.weird.com/pub/local/
```

The pcl-cvs package it's referring to is very handy, and we discuss it in Chapter 10.

The src/ and lib/ subdirectories contain the bulk of the CVS source code, which involves the CVS internals. The main data structures and commands are implemented in src/, whereas lib/ contains small code modules of general utility that CVS uses.

The man/ subdirectory contains the CVS man pages (intended for the Unix online manual system). When you ran **make install**, they were incorporated into your Unix system's regular man pages, so you can type

```
yarkon$ man cvs
```

and get a rather terse introduction and subcommand reference to CVS. Although useful as a quick reference, the man pages may not be as up to date or complete as the Cederqvist manual (see the next section); however, if all you have are the man pages, you can usually sort things out.

The Cederqvist Manual

That leaves the doc/ subdirectory, whose most important inhabitant is the famed Cederqvist. These days, it's probably a stretch to call it "the Cederqvist." Although Per Cederqvist (of Signum Support, Linkoping Sweden, **www.signum.se**) wrote the first version around 1992, it has been updated since then by many other people. For example, when contributors add a new feature to CVS, they usually also document it in the Cederqvist.

The Cederqvist manual is written in the Texinfo format, which is used by the GNU project because it's relatively easy to produce both online and printed output from it (in Info and PostScript formats, respectively). The Texinfo master file is doc/cvs.texinfo, but CVS distributions come with the Info and PostScript already generated, so you don't have to worry about running any Texinfo tools yourself.

Although the Cederqvist can be used as an introduction and tutorial, it is probably most useful as a reference document. For that reason, most people browse it online instead of printing it out (although the PostScript file is doc/cvs.ps, for those with paper to spare). If this is the first time you've installed CVS on your system, you'll have to take an extra step to make sure the manual is accessible online.

The Info files (doc/cvs.info, doc/cvs.info-1, doc/cvs.info-2, and so on) were installed for you when you ran **make install**. Although the files were copied into the system's Info tree, you might still have to add a line for CVS to the Info table of contents, the "Top" node. (This is necessary only if this is the first time CVS has been installed on your system; otherwise, the entry from previous installations should already be in the table of contents.)

If you've added new Info documentation before, you might be familiar with the process. First, figure out where the Info pages were installed. If you used the default installation (in /usr/local/), then the Info files are /usr/local/info/cvs.info*. If you installed using

```
yarkon$ ./configure —prefix=/usr
```

the files ended up as /usr/info/cvs.*. After you locate the files, you'll need to add a line for CVS to the Info table of contents, which is in a file named dir in that directory (so in the latter case, it would be /usr/info/dir). If you don't have root access, ask your system administrator to do it. Here is an excerpt from dir before the reference to CVS documentation was added:

```
* Bison: (bison).         The Bison parser generator.
* Cpp: (cpp).             The GNU C preprocessor.
* Flex: (flex).           A fast scanner generator
```

And here is the same region of dir afterward:

```
* Bison: (bison).          The Bison parser generator.
* Cpp: (cpp).              The GNU C preprocessor.
* Cvs: (cvs).              Concurrent Versions System
* Flex: (flex).            A fast scanner generator
```

The format of the line is very important. You must include the asterisk, spaces, and colon in "*** Cvs:** " and the parentheses and period in "**(cvs).**" after it. If any of these elements is missing, the Info dir format will be corrupt, and you'll be unable to read the Cederqvist.

Once the manual is installed and referred to from the table of contents, you can read it with any Info-compatible browser. The ones most likely to be installed on a typical Unix system are either the command-line Info reader, which can be invoked this way if you want to go straight to the CVS pages

```
yarkon$ info cvs
```

and the one within Emacs, which is invoked by typing

```
M-x info
```

or:

```
C-h i
```

Take whatever time is necessary to get the Cederqvist set up properly on your system when you install CVS; it will pay off many times down the road when you need to look something up.

Other Sources of Information

In addition to the Cederqvist, the FAQ, and the other files in the distribution itself, there are Internet resources devoted to CVS. If you're going to administer a CVS server, you'll probably want to join the info-cvs mailing list. To subscribe, send email to **info-cvs-request@gnu.org** (the list itself is **info-cvs@gnu.org**). Traffic can be medium to heavy, around 10 to 20 emails a day, most of them questions seeking answers. The majority of these can be deleted without reading (unless you want to help people by answering their questions, which is always nice), but every now and then someone will announce the discovery of a bug or announce a patch that implements some feature you've been wanting.

You can also join the formal bug report mailing list, which includes every bug report sent in. This probably isn't necessary, unless you intend to help fix the bugs, which would be great, or you're terrifically paranoid and want to know about every problem other people find with CVS. If you do want to join, send email to **bug-cvs-request@gnu.org**.

There's also a Usenet newsgroup, **comp.software.config-mgmt**, which is about version control and configuration management systems in general, in which there is a fair amount of discussion about CVS.

Finally, there are at least three Web sites devoted to CVS. Cyclic Software's **http://cvshome.org/** has been CVS's informal home site for a few years and probably will continue to be for the foreseeable future. Cyclic Software also provides server space and Net access for the repository where the CVS sources are kept. The Cyclic Web pages contain comprehensive links to experimental patches for CVS, third-party tools that work with CVS, documentation, mailing list archives, and just about everything else. If you can't find what you need in the distribution, **http://cvshome.org/** is the place to start looking.

Two other good sites are Pascal Molli's **www.loria.fr/~molli/cvs-index.html** and Sean Dreilinger's **http://durak.org/cvswebsites/**. The biggest attraction at Molli's site is, of course, the FAQ, but it also has links to CVS-related tools and mailing list archives. Dreilinger's site specializes in information about using CVS to manage Web documents and also has a CVS-specific search engine.

Starting a Repository

Once the CVS executable is installed on your system, you can start using it right away as a client to access remote repositories, following the procedures we described in Chapter 2. However, if you want to serve revisions from your machine, you have to create a repository there. The command to do that is

```
yarkon$ cvs -d /usr/local/newrepos init
```

where /usr/local/newrepos is a path to wherever you want the repository to be. (Of course, you must have write permission to that location, which might imply running the command as the root user.) It might seem somewhat counterintuitive that the location of the new repository is specified before the **init** subcommand instead of after it, but by using the **-d** option, it stays consistent with other CVS commands.

The command will return silently after it is run. Let's examine the new directory:

```
yarkon$ ls -ld /usr/local/newrepos
drwxrwxr-x   3 root     root         1024 Jun 24 17:59 /usr/local/newrepos/
yarkon$ cd /usr/local/newrepos
yarkon$ ls
CVSROOT
yarkon$ cd CVSROOT
yarkon$ ls
checkoutlist     config,v        history     notify     taginfo,v
checkoutlist,v   cvswrappers     loginfo     notify,v   verifymsg
```

```
commitinfo       cvswrappers,v   loginfo,v   rcsinfo    verifymsg,v
commitinfo,v     editinfo        modules     rcsinfo,v
config           editinfo,v      modules,v   taginfo

yarkon$
```

The single subdirectory in the new repository—CVSROOT/—contains various administrative files that control CVS's behavior. Later on, we examine those files one by one; for now, the goal is just to get the repository working. In this case, "working" means users can import, check out, update, and commit projects.

Note

*Don't confuse the **CVSROOT** environment variable introduced in Chapter 2 with this CVSROOT subdirectory in the repository. They are unrelated; it is an unfortunate coincidence that they share the same name. The former is a way for users to avoid having to type **-d <repository-location>** every time they use CVS; the latter is the administrative subdirectory of a repository.*

Once the repository is created, you must take care of its permissions. CVS does not require any particular, standardized permission or file ownership scheme; it merely needs write access to the repository. However—partly for security reasons, but mainly for your own sanity as an administrator—we *highly* recommend that you take the following steps:

1. Add a Unix group "cvs" to your system. Any users who need to access the repository should be in this group. For example, here's the relevant line from a typical machine's /etc/group file:

   ```
   cvs:*:105:mosheb,kfogel,anonymous,jrandom
   ```

2. Make the repository's group ownership and permissions reflect this new group:

   ```
   yarkon$ cd /usr/local/newrepos
   yarkon$ chgrp -R cvs .
   yarkon$ chmod ug+rwx . CVSROOT
   ```

Now any of the users listed in that group can start a project by running **cvs import**, as described in Chapter 2. **Checkout**, **update**, and **commit** should work as well. Users can also reach the repository from remote locations by using the **:ext:** method, assuming that they have **rsh** or **ssh** access to the repository machine. (You might have noticed that the **chgrp** and **chmod** commands in that example gave *write* access to a user named "anonymous," which is not what one would expect. The reason is that even anonymous, read-only repository users need system-level write access, so that their CVS processes can create temporary lockfiles inside the repository. CVS enforces the "read-only" restriction of anonymous access not through Unix file permissions, but by other means, which we cover shortly.)

If your repository is intended to serve projects to the general public, where contributors won't necessarily have accounts on the repository machine, you should set up the password-authenticating server now. It's necessary for anonymous read-only access, and it's also probably the easiest way to grant commit access to certain people without giving them full accounts on the machine.

The Password-Authenticating Server

Before running through the steps needed to set up the password server, let's examine how such connections work in the abstract. When a remote CVS client uses the **:pserver:** method to connect to a repository, the client is actually contacting a specific port number on the server machine—specifically, port number 2401 (which is 49 squared, if you like that sort of thing). Port 2401 is the designated default port for the CVS pserver, although one could arrange for a different port to be used as long as both client and server agree on it.

The CVS server is not actually waiting for connections at that port—the server won't get started up until a connection actually arrives. Instead, the Unix inetd (Internet daemon) program is listening on that port and needs to know that when it receives a connection request there, it should start up the CVS server and connect it to the incoming client.

This is accomplished by modifying inetd's configuration files: /etc/services and /etc/inetd.conf. The services file maps raw port numbers to service names, and then inetd.conf tells inetd what to do for a given service name.

First, put a line like this into /etc/services (after checking to make sure it isn't already there):

```
cvspserver 2401/tcp
```

Then in /etc/inetd.conf, put this:

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

(In the actual file, this should be all one long line, with no backslash.) If your system uses tcpwrappers, you might want to use something like this instead:

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

Now, restart inetd so it notices the changes to its configuration files. (If you don't know how to restart the daemon, just reboot the machine—that will work, too.)

That's enough to permit connections, but you'll also want to set up special CVS passwords—separate from the users' regular login passwords—so people can access the repository without compromising overall system security.

The CVS password file is CVSROOT/passwd in the repository. It was not created by default when you ran **cvs init**, because CVS doesn't know for sure that you'll be using pserver. Even if the password file has been created, CVS has no way of knowing what usernames and passwords to create. So, you have to create one yourself; here's a sample CVSRoot/passwd file:

```
kfogel:rKa5jzULzmhOo

mosheb:tGX1fS8sun6rY:pubcvs
anonymous:XR4EZcEs0szik
```

The format is as simple as it looks. Each line is:

```
<USERNAME>:<ENCRYPTED_PASSWORD>:<OPTIONAL_SYSTEM_USERNAME>
```

The extra colon followed by an optional system username tells CVS that connections authenticated with **USERNAME** should run as the system account **SYSTEM_USERNAME**. In other words, CVS session is able to do things in the repository that only someone logged in as **SYSTEM_USERNAME** can do.

If no system username is given, **USERNAME** must match an actual login account name on the system, and the session will run with that user's permissions. In either case, the encrypted password should not be the same as the user's actual login password. It should be an independent password used only for CVS pserver connections.

The password is encrypted using the same algorithm as the standard Unix system passwords stored in /etc/passwd. You might be wondering at this point how one acquires an encrypted version of a password. For Unix system passwords, the **passwd** command takes care of the encryption in /etc/passwd for you. Unfortunately, there is no corresponding **cvs passwd** command (it has been proposed several times, but no one's gotten around to writing it—perhaps you'll do it?).

This is an inconvenience, but only a slight one. If nothing else, you can always temporarily change a regular user's system password using **passwd**, cut and paste the encrypted text from /etc/passwd into CVSROOT/passwd, and then restore the old password.

Note

On some systems, the encrypted passwords are found in /etc/shadow and are readable only by root.

That scheme is workable but rather cumbersome. It would be much easier to have a command-line utility that takes a plain text password as its argument and outputs the encrypted version. Here is such a tool, written in Perl:

```
#!/usr/bin/perl
```

```
srand (time());
my $randletter = "(int (rand (26)) + (int (rand (1) + .5) % 2 ? 65 : 97))";
my $salt = sprintf ("%c%c", eval $randletter, eval $randletter);
my $plaintext = shift;
my $crypttext = crypt ($plaintext, $salt);

print "${crypttext}\n";
```

A good location for the preceding is in /usr/local/bin/cryptout.pl:

```
yarkon$ ls -l /usr/local/bin/cryptout.pl

-rwxr-xr-x   1   root   root   265  Jun 14 20:41 /usr/local/bin/cryptout.pl
yarkon$ cryptout.pl "some text"
sB3A79YDX5L4s

yarkon$
```

If we took the output of this example and used it to create the following entry in CVSROOT/passwd

```
jrandom:sB3A79YDX5L4s:craig
```

then someone could connect to the repository with the following command:

```
remote$ cvs -d :pserver:jrandom@yarkon.red-bean.com:/usr/local/newrepos login
```

They could then type "some text" as their password and thereafter be able to execute CVS commands with the same access privileges as the system user "craig."

If someone attempts to authenticate with a username and password that don't appear in CVSROOT/passwd, CVS will check to see if that username and password are present in /etc/passwd. If they are (and if the password matches, of course), CVS will grant access. It behaves this way for the administrator's convenience, so that separate CVSROOT/passwd entries don't have to be set up for regular system users. However, this behavior is also a security hole, because it means that if one of those users does connect to the CVS server, that user's regular login password will have crossed over the network in cleartext, potentially vulnerable to the eyes of password sniffers. A bit later, you learn how to turn off this "fallback" behavior, so that CVS consults only its own passwd file. However, whether you leave it on or off, you should probably force any CVS users who also have login accounts to maintain different passwords for the two functions.

Although the passwd file authenticates for the whole repository, with a little extra work you can still use it to grant project-specific access. Here's one method. Suppose you want to grant some remote developers access to project "foo," and others access to project "bar," and

you don't want developers from one project to have **commit** access to the other. You can accomplish this by creating project-specific user accounts and groups on the system and then mapping to those accounts in the CVSROOT/passwd file. Here's the relevant excerpt from /etc/passwd

```
cvs-foo:*:600:600:Public CVS Account for Project Foo:/usr/local/cvs:/bin/false
cvs-bar:*:601:601:Public CVS Account for Project Bar:/usr/local/cvs:/bin/false
```

and from /etc/group

```
cvs-foo:*:600:cvs-foo
cvs-bar:*:601:cvs-bar
```

and, finally, CVSROOT/passwd:

```
kcunderh:rKa5jzULzmhOo:cvs-foo
jmankoff:tGX1fS8sun6rY:cvs-foo
brebard:cAXVPNZN6uFH2:cvs-foo
xwang:qp5lsf7nzRzfs:cvs-foo
dstone:JDNNF6HeX/yLw:cvs-bar
twp:glUHEM8KhcbO6:cvs-bar
ffranklin:cG6/6yXbS9BHI:cvs-bar
yyang:YoEqcCeCUq1vQ:cvs-bar
```

Some of the CVS usernames map onto the system user account cvs-foo and some onto cvs-bar. Because CVS runs under the user ID of the system account, you just have to make sure that only the appropriate users and groups can write to the relevant parts of the repository. If you just make sure that the user accounts are locked down pretty tight (no valid login password, /bin/false as the shell), then this system is reasonably secure (but see information later in this chapter about CVSROOT permissions!). Also, CVS does record changes and log messages under the CVS username, not the system username, so you can still tell who is responsible for a given change.

Anonymous Access Via the Password-Authenticating Server

So far, we've seen only how to use the password-authenticating server to grant normal full access to the repository (although, admittedly, one can restrict that access through carefully arranged Unix file permissions). Turning this into anonymous, read-only access is a simple step: You just have to add a new file, or possibly two, in CVSROOT/. The files' names are "readers" and "writers"—the former containing a list of usernames for users who can only read the repository, the latter for users who can read and write.

If you list a username in CVSROOT/readers, that user will have only read access to all projects in the repository. If you list a username in CVSROOT/writers, that user will have write access, and every pserver user *not* listed in writers will have read-only access. (In other

words, if the writers file exists at all, it implies read-only access for all those not listed in it.) If the same username is listed in both files, CVS resolves the conflict in the more conservative way: The user will have read-only access.

The format of the files is very simple: one user per line (don't forget to put a newline after the last user). Here is a sample readers file:

```
anonymous
splotnik
guest
jbrowse
```

Note that the files apply to CVS usernames, not system usernames. If you use user aliasing in the CVSROOT/passwd file (putting a system username after a second colon), the leftmost username is the one to list in a readers or writers file.

Formally, here's how the server works in deciding whether to grant read-only or read-write access. If a readers file exists and this user is listed in it, then she gets read-only access. If a writers file exists and this user is not listed in it, then she also gets read-only access (this is true even if a readers file exists but that person is not listed there). If that person is listed in both, she gets read-only access. In all other cases, that person gets full read-write access.

Thus, a typical repository with anonymous CVS access has this (or something like it) in CVSROOT/passwd

```
anonymous:XR4EZcEsOszik
```

this (or something like it) in /etc/passwd

```
anonymous:!:1729:105:Anonymous CVS User:/usr/local/newrepos:/bin/false
```

and this in CVSROOT/readers:

```
anonymous
```

And, of course, the aforementioned setup in /etc/services and /etc/inetd.conf. That's all there is to it!

Note that some older Unix systems don't support usernames longer than eight characters. One way to get around this is to call the user "anon" instead of "anonymous" in CVSROOT/passwd and in the system files, because people often assume that anon is short for anonymous, anyway. However, it is probably better to put something like this into the CVSROOT/passwd file

```
anonymous:XR4EZcEsOszik:cvsanon
```

(and then, of course, use "cvsanon" in the system files). That way, you can publish a repository address that uses "anonymous," which is more or less standard now. People accessing the repository with

```
cvs -d :pserver:anonymous@cvs.foobar.com:/usr/local/newrepos (etc...)
```

would actually run on the server as cvsanon (or whatever). However, they wouldn't need to know or care about how things are set up on the server side—they'd see only the published address.

Repository Structure Explained in Detail

The new repository still has no projects in it. Let's re-run the initial import from Chapter 2, watching what happens to the repository. (For simplicity's sake, all commands will assume that the **CVSROOT** environment variable has been set to /usr/local/newrepos, so there's no need to specify the repository with **-d** on imports and checkouts.)

```
yarkon$ ls /usr/local/newrepos
CVSROOT/
yarkon$ pwd
/home/jrandom/src/
yarkon$ ls
myproj/
yarkon$ cd myproj
yarkon$ cvs import -m "initial import into CVS" myproj jrandom start
N myproj/README.txt
N myproj/hello.c
cvs import: Importing /usr/local/newrepos/myproj/a-subdir
N myproj/a-subdir/whatever.c
cvs import: Importing /usr/local/newrepos/myproj/a-subdir/subsubdir
N myproj/a-subdir/subsubdir/fish.c
cvs import: Importing /usr/local/newrepos/myproj/b-subdir
N myproj/b-subdir/random.c

No conflicts created by this import

yarkon$ ls /usr/local/newrepos
CVSROOT/  myproj/
yarkon$ cd /usr/local/newrepos/myproj
yarkon$ ls
README.txt,v  a-subdir/     b-subdir/     hello.c,v
yarkon$ cd a-subdir
yarkon$ ls
subsubdir/    whatever.c,v
yarkon$ cd ..

yarkon$
```

Before the import, the repository contained only its administrative area, CVSROOT. After the import, a new directory—myproj—appeared. The files and subdirectories inside that new directory look suspiciously like the project we imported, except that the files have the suffix ",v." These are RCS-format version control files (the ",v" stands for "version"), and they are the backbone of the repository. Each RCS file stores the revision history of its corresponding file in the project, including all branches and tags.

You do not need to know any of the RCS format to use CVS (although there is an excellent write-up included with the source distribution in doc/RCSFILES). However, a basic understanding of the format can be of immense help in troubleshooting CVS problems, so we'll take a brief peek into one of the files, hello.c,v. Here are its contents:

```
head     1.1;
branch   1.1.1;
access   ;
symbols  start:1.1.1.1 jrandom:1.1.1;
locks    ; strict;
comment  @ * @;

1.1
date     99.06.20.17.47.26;  author jrandom;  state Exp;
branches 1.1.1.1;
next;

1.1.1.1
date     99.06.20.17.47.26;  author jrandom;  state Exp;
branches ;
next;

desc
@@

1.1
log
@Initial revision
@
text
@#include <stdio.h>

void
main ()
{
  printf ("Hello, world!\n");
}
@
```

```
1.1.1.1
log
@initial import into CVS
@
text
@@
```

Whew! Most of that you can ignore; don't worry about the relationship between 1.1 and 1.1.1.1, for example, or the implied 1.1.1 branch. They aren't really significant from a user's or even an administrator's point of view. What you should try to understand is the overall format. At the top is a collection of header fields:

```
head     1.1;
branch   1.1.1;
access   ;
symbols  start:1.1.1.1 jrandom:1.1.1;
locks    ; strict;
comment  @ * @;
```

Farther down in the file are groups of meta-information about each revision (but still not showing the contents of that revision), such as:

```
1.1
date     99.06.20.17.47.26;  author jrandom;  state Exp;
branches 1.1.1.1;
next     ;
```

And finally, the log message and text of an actual revision:

```
1.1
log
@Initial revision
@
text
@#include <stdio.h>

void
main ()
{
  printf ("Hello, world!\n");
}
@

1.1.1.1
log
```

```
@initial import into CVS
@
text
@@
```

If you look closely, you'll see that the first revision's contents are stored under the heading 1.1; however, the log message there is "Initial revision," whereas the log message we actually used at import time was "initial import into CVS," which appears farther down, under revision 1.1.1.1. You don't need to worry about this discrepancy right now. This happens because imports are a special circumstance: In order for repeated imports into the same project to have a useful effect, **import** actually places the initial revision on both the main trunk *and* on a special branch. (The reasons for this will become clearer when we look at vendor branches in Chapter 4). For now, you can treat 1.1 and 1.1.1.1 as the same thing.

The file becomes even more revealing after we commit the first modification to hello.c:

```
yarkon$ cvs -Q co myproj
yarkon$ cd myproj
yarkon$ emacs hello.c
    (make some changes to the file)

yarkon$ cvs ci -m "print goodbye too"
cvs commit: Examining .
cvs commit: Examining a-subdir
cvs commit: Examining a-subdir/subsubdir
cvs commit: Examining b-subdir
Checking in hello.c;
/usr/local/newrepos/myproj/hello.c,v  <--  hello.c
new revision: 1.2; previous revision: 1.1
done
```

If you look at hello.c,v in the repository now, you can see the effect of the commit:

```
head  1.2;
access;
symbols
      start:1.1.1.1 jrandom:1.1.1;
locks; strict;
comment   @ * @;

1.2
date   99.06.21.01.49.40;   author jrandom;   state Exp;
branches;
next   1.1;
```

```
1.1
date   99.06.20.17.47.26;   author jrandom;   state Exp;
branches
       1.1.1.1;
next   ;

1.1.1.1
date   99.06.20.17.47.26;   author jrandom;   state Exp;
branches;
next   ;

desc
@@

1.2
log
@print goodbye too
@
text
@#include <stdio.h>

void
main ()
{
  printf ("Hello, world!\n");
  printf ("Goodbye, world!\n");
}
@

1.1
log
@Initial revision
@
text
@d7 1
@

1.1.1.1
log
@initial import into CVS
@
text
@@
```

Now the full contents of revision 1.2 are stored in the file, and the text for revision 1.1 has been replaced with the cryptic formula:

```
d7 1
```

The **d7 1** is a diff notation meaning "starting at line 7, delete 1 line." In other words, to derive revision 1.1, delete line 7 from revision 1.2! Try working through it yourself. You'll see that it does indeed produce revision 1.1—it simply does away with the line we added to the file.

This demonstrates the basic principle of RCS format: It stores only the differences between revisions, thereby saving a lot of space compared with storing each revision in full. To go backwards from the most recent revision to the previous one, it patches the later revision using the stored diff. Of course, this means that the further back you go in the revision history, the more patch operations must be performed. (For example, if the file is on revision 1.7 and CVS is asked to retrieve revision 1.4, it has to produce 1.6 by patching backwards from 1.7, then 1.5 by patching 1.6, then 1.4 by patching 1.5.) Fortunately, old revisions are also the ones least often retrieved, so the RCS system works out pretty well in practice: The more recent the revision, the cheaper it is to obtain.

As for the header information at the top of the file, you don't need to know what all of it means. However, the effects of certain operations show up very clearly in the headers, and a passing familiarity with them might prove useful.

When committing a new revision on the trunk, the "head" label is updated (note how it became 1.2 in the preceding example, when the second revision to hello.c was committed). By adding a file as binary or when tagging it, those operations are recorded in the headers as well. As an example, we'll add foo.jpg as a binary file and then tag it a couple of times:

```
yarkon$ cvs add -kb foo.jpg
cvs add: scheduling file 'foo.jpg' for addition
cvs add: use 'cvs commit' to add this file permanently
yarkon$ cvs -q commit -m "added a random image; ask jrandom@red-bean.com why"
RCS file: /usr/local/newrepos/myproj/foo.jpg,v
done
Checking in foo.jpg;
/usr/local/newrepos/myproj/foo.jpg,v  <--  foo.jpg
initial revision: 1.1
done
yarkon$ cvs tag some_random_tag foo.jpg
T foo.jpg
yarkon$ cvs tag ANOTHER-TAG foo.jpg
T foo.jpg
yarkon$
```

Now examine the header section of foo.jpg,v in the repository:

```
head   1.1;
access;
symbols
```

```
      ANOTHER-TAG:1.1
      some_random_tag:1.1;
locks; strict;
comment   @# @;
expand     @b@;
```

Notice the **b** in the **expand** line at the end—it's due to our having used the **-kb** flag when adding the file and means the file won't experience any keyword or newline expansions, which would normally occur during checkouts and updates if it were a regular text file. The tags appear in the **symbols** section, one tag per line—both of them are attached to the first revision, because that's what was tagged both times. This also helps explain why tag names can contain only letters, numbers, hyphens, and underscores. If the tag itself contained colons or dots, the RCS file's record of it might be ambiguous, because there would be no way to find the textual boundary between the tag and the revision to which it is attached.

RCS Format Always Quotes @ Signs

The @ symbol is used as a field delimiter in RCS files, which means that if one appears in the text of a file or in a log message, it must be quoted (otherwise, CVS would incorrectly interpret it as marking the end of that field). It is quoted by doubling—that is, CVS always interprets @@ as "literal @ sign," never as "end of current field." When we committed foo.jpg, the log message was

```
"added a random image; ask jrandom@red-bean.com why"
```

which is stored in foo.jpg,v like this:

```
1.1
log
@added a random image; ask jrandom@@red-bean.com why
@
```

The @ sign in **jrandom@@red-bean.com** will be automatically unquoted whenever CVS retrieves the log message:

```
yarkon$ cvs log foo.jpg
RCS file: /usr/local/newrepos/myproj/foo.jpg,v
Working file: foo.jpg
head: 1.1
branch:
locks: strict
access list:
symbolic names:
      ANOTHER-TAG: 1.1
      some_random_tag: 1.1
```

```
keyword substitution: b
total revisions: 1;       selected revisions: 1
description:
----------------------------
revision 1.1
date: 2001/06/21 02:56:18;  author: jrandom;  state: Exp;
added a random image; ask jrandom@red-bean.com why
=============================================================================

yarkon$
```

The only reason you should care is that if you ever find yourself hand-editing RCS files (a rare circumstance, but not unheard of), you must remember to use double @ signs in revision contents and log messages. If you don't, the RCS file will be corrupt and will probably exhibit strange and undesirable behaviors.

Speaking of hand-editing RCS files, don't be fooled by the permissions in the repository:

```
yarkon$ ls -l
total 6
-r--r--r--   1 jrandom   users         410 Jun 20 12:47 README.txt,v
drwxrwxr-x   3 jrandom   users        1024 Jun 20 21:56 a-subdir/
drwxrwxr-x   2 jrandom   users        1024 Jun 20 21:56 b-subdir/
-r--r--r--   1 jrandom   users         937 Jun 20 21:56 foo.jpg,v
-r--r--r--   1 jrandom   users         564 Jun 20 21:11 hello.c,v

yarkon$
```

Although the files appear to be read-only for everyone, the directory permissions must also be taken into account:

```
yarkon$ ls -ld .
drwxrwxr-x   4 jrandom   users        1024 Jun 20 22:16 ./
yarkon$
```

The myproj/ directory itself—and its subdirectories—are all writable by the owner (jrandom) and the group (users). This means that CVS (running as jrandom or as anyone in the users group) can create and delete files in those directories, even if it can't directly edit them. CVS edits an RCS file by making a separate copy of it, so you should also make all of your changes in a temporary copy and then replace the existing RCS file with the new one. (Please don't ask why the files themselves are read-only—there are historical reasons for that, having to do with the way RCS works when run as a standalone program.)

Incidentally, having the files' group be "users" is probably not what you want, considering that the top-level directory of the repository was explicitly assigned group "cvs".You can correct the problem by running this command inside the repository:

```
yarkon$ cd /usr/local/newrepos
yarkon$ chgrp -R cvs myproj
```

Unfortunately, the usual Unix file-creation rules govern which group is assigned to new files that appear in the repository, so once in a while you might need to run **chgrp** or **chmod** on certain files or directories in the repository. There are no hard and fast rules about how you should structure repository permissions; it just depends on who is working on which projects.

What Happens When You Remove a File

When you remove a file from a project, it doesn't just disappear. CVS must be able to retrieve such files when you request an old snapshot of the project. Instead, the file gets put in the "Attic" directory:

```
yarkon$ pwd
/home/jrandom/src/myproj
yarkon$ ls /usr/local/newrepos/myproj/
README.txt,v  a-subdir/     b-subdir/     foo.jpg,v   hello.c,v
yarkon$ rm foo.jpg
yarkon$ cvs rm foo.jpg
cvs remove: scheduling 'foo.jpg' for removal
cvs remove: use 'cvs commit' to remove this file permanently
yarkon$ cvs ci -m "Removed foo.jpg" foo.jpg
Removing foo.jpg;
/usr/local/newrepos/myproj/foo.jpg,v  <--  foo.jpg
new revision: delete; previous revision: 1.1
done
yarkon$ cd /usr/local/newrepos/myproj/
yarkon$ ls
Attic/      README.txt,v  a-subdir/     b-subdir/   hello.c,v
yarkon$ cd Attic
yarkon$ ls
foo.jpg,v
yarkon$
```

In each repository directory of a project, the presence of an Attic/ subdirectory means that at least one file has been removed from that directory (this means that you shouldn't use directories named Attic in your projects). CVS doesn't merely move the RCS file into Attic/, however; it also commits a new revision into the file, with a special revision state of "dead." Here's the relevant section from Attic/foo.jpg,v:

```
1.2
date   99.06.21.03.38.07;   author jrandom;   state dead;
branches;
next       1.1;
```

If the file is later re-activated, CVS keeps track of the fact that it was dead at some point in the past and is now alive again. This means that if you want to restore a removed file, you can't just take it out of the Attic/ and put it back into the project. Instead, you have to do something like this in a working copy:

```
yarkon$ pwd
/home/jrandom/src/myproj
yarkon$ cvs -Q update -p -r 1.1 foo.jpg > foo.jpg
yarkon$ ls
CVS/       README.txt   a-subdir/   b-subdir/   foo.jpg     hello.c
yarkon$ cvs add -kb foo.jpg
cvs add: re-adding file foo.jpg (in place of dead revision 1.2)
cvs add: use 'cvs commit' to add this file permanently
yarkon$ cvs ci -m "revived jpg image" foo.jpg
Checking in foo.jpg;
/usr/local/newrepos/myproj/foo.jpg,v  <-- foo.jpg
new revision: 1.3; previous revision: 1.2
done
yarkon$ cd /usr/local/newrepos/myproj/
yarkon$ ls
Attic/          a-subdir/     foo.jpg,v
README.txt,v  b-subdir/     hello.c,v
yarkon$ ls Attic/
yarkon$
```

There's a lot more to know about RCS format, but this is sufficient for a CVS administrator to be able to maintain a repository. You will probably never have to actually edit an RCS file; you'll usually just have to tweak file permissions in the repository (at least, this has been our experience). Nevertheless, when CVS starts behaving truly weirdly (rare, but not completely outside the realm of possibility), you should first look inside the RCS files to figure out what's wrong.

The CVSROOT/ Administrative Directory

The files in newrepos/CVSROOT/ are not part of any project, but they are used to control CVS's behavior in the repository. The best way to edit those files is to check out a working copy of CVSROOT, just as with a regular project:

```
yarkon$ cvs co CVSROOT
cvs checkout: Updating CVSROOT
U CVSROOT/checkoutlist
U CVSROOT/commitinfo
U CVSROOT/config
U CVSROOT/cvswrappers
U CVSROOT/editinfo
U CVSROOT/loginfo
```

```
U CVSROOT/modules
U CVSROOT/notify
U CVSROOT/rcsinfo
U CVSROOT/taginfo
U CVSROOT/verifymsg
yarkon$
```

We'll address the files in their approximate order of importance. Note that each of the files comes with an explanatory comment at the beginning (the comment convention is the same across all of them: A "#" sign at the beginning of the line signifies a comment, and CVS ignores such lines when parsing the files). Remember that any change you make to the administrative files in your checked-out working copy won't affect CVS's behavior until you commit the changes.

Tip

In today's world of crackers and malicious users, it is advisable to be sure that the Unix-level permissions on CVSROOT are different from permissions elsewhere in the repository, in order to have fine-grained control over who can commit changes to CVSROOT. As you'll see a little later, being able to modify the files in CVSROOT essentially gives any CVS user—even remote ones—the ability to run arbitrary commands on the repository machine.

The config File

The config file allows you to configure certain global behavioral parameters. It follows a very strict format

```
PARAMETER=VALUE
(etc)
```

with no extra spaces allowed. For example, here is a possible config file:

```
SystemAuth=yes
TopLevelAdmin=no
PreservePermissions=no
```

(An absent entry is equivalent to "no.") The **SystemAuth** parameter indicates whether CVS should look in the system passwd file if it fails to find a given username in the CVSROOT/passwd file. CVS distributions are shipped with this set to "no" to be conservative about your system's security.

TopLevelAdmin tells CVS whether to make a sibling CVS/ directory when it checks out a working copy. This CVS/ directory would not be *inside* the working copy, but next to it. It might be convenient to turn this on if you tend (and your repository's users tend) to check out many different projects from the same repository. Otherwise, you should leave it off, because it can be disconcerting to see an extra CVS/ directory appear where you don't expect it.

PreservePermissions governs whether to preserve file permissions and similar metadata in the revision history. This is a somewhat obscure feature that probably isn't worth describing in detail. See the node "Special Files" in the Cederqvist if you're interested.

Tip

"Node" is Texinfo-speak for a particular location within an Info document. To go to a node while reading Info, just type "g" followed by the name of the node, from anywhere inside the document.

LockDir is also a rarely used feature. In special circumstances, you might want to tell CVS to create its lockfiles somewhere other than directly in the project subdirectories, in order to avoid permission problems. These lockfiles serialize multiple operations on the same repository directory simultaneously. Generally, you never need to worry about them, but sometimes users might have trouble updating or checking out from a repository directory because they're unable to create a lockfile. (This can happen because even on read-only operations, CVS needs to create a lockfile to avoid situations where it could end up reading while another invocation of CVS is writing.) The usual fix for this is to change repository permissions, but when that's not feasible, the **LockDir** parameter can come in handy.

There are no other parameters at this time, but future versions of CVS might add new ones. You should always check the Cederqvist manual or the distribution config file itself for updates.

The modules File

In modules, you can define aliases and alternate groupings for projects in the repository. The most basic module line is of the form:

```
MODULE_NAME   DIRECTORY_IN_REPOSITORY
```

Here's an example:

```
mp    myproj
asub  myproj/a-subdir
```

The paths given on the right are relative to the top of the repository. These paths give developers an alternate name by which to check out a project or a portion of a project:

```
yarkon$ cvs co mp
cvs checkout: Updating mp
U mp/README.txt
U mp/foo.jpg
U mp/hello.c
cvs checkout: Updating mp/a-subdir
U mp/a-subdir/whatever.c
```

```
cvs checkout: Updating mp/a-subdir/subsubdir
U mp/a-subdir/subsubdir/fish.c
cvs checkout: Updating mp/b-subdir
U mp/b-subdir/random.c
```

or

```
yarkon$ cvs -d /usr/local/newrepos/ co asub
cvs checkout: Updating asub
U asub/whatever.c
cvs checkout: Updating asub/subsubdir
U asub/subsubdir/fish.c
```

Notice how in both cases the module's name became the name of the directory created for the working copy. In the case of asub, CVS didn't even bother with the intermediate myproj/ directory, but created a top-level asub/ instead, even though it came from myproj/a-subdir in the repository. Updates, commits, and all other CVS commands will behave normally in those working copies; their names are the only things unusual about them.

By putting file names after the directory name, you can define a module consisting of just some of the files in a given repository directory. For example

```
readme  myproj  README.txt
```

and

```
no-readme  myproj  hello.c  foo.jpg
```

permit the following checkouts, respectively:

```
yarkon$ cvs -q co readme
U readme/README.txt
yarkon$ cvs -q co no-readme
U no-readme/hello.c
U no-readme/foo.jpg
yarkon$
```

You can define a module that will include multiple repository directories by using the **-a** (for "alias") flag, but note that the directories will get them checked out under their original names. For example, this line

```
twoproj  -a  myproj  yourproj
```

would allow you to do this (assuming that both myproj/ and yourproj/ are in the repository):

```
yarkon$ cvs co twoproj
U myproj/README.txt
U myproj/foo.jpg
U myproj/hello.c
U myproj/a-subdir/whatever.c
U myproj/a-subdir/subsubdir/fish.c
U myproj/b-subdir/random.c
U yourproj/README
U yourproj/foo.c
U yourproj/some-subdir/file1.c
U yourproj/some-subdir/file2.c
U yourproj/some-subdir/another-subdir/blah.c
```

The name "twoproj" was a convenient handle to pull in both projects, but it didn't affect the names of the working copies. (There is no requirement that alias modules refer to multiple directories, by the way; we could have omitted twoproj, in which case myproj would still have been checked out under the name "myproj.")

Modules can even refer to other modules, by prefixing them with an ampersand:

```
mp    myproj
asub  myproj/a-subdir
twoproj -a myproj yourproj
tp  &twoproj
```

Doing a checkout of tp has exactly the same result as the checkout of twoproj.

There are a few other tricks you can do with modules, most of them less frequently used than the ones just presented. See the node "Modules" in the Cederqvist for information about them.

The commitinfo and loginfo Files

Most of the other administrative files provide programmatic "hooks" into various parts of the commit process (for example, the ability to validate log messages or file states before permitting the commit, or the ability to notify a group of developers whenever a commit happens in a certain directory of the repository).

The files generally share a common syntax. Each line is of the form:

```
REGULAR_EXPRESSION    PROGRAM_TO_RUN
```

The regular expression will be tested against the directory into which the commit is taking place (with the directory name relative to the top of the repository). If it matches, the designated program will be run. The program will be passed the names of each of the files in the commit; it can do whatever it likes with those names, including opening up the files and

examining their contents. If the program returns with a nonzero exit status, the commit is prevented from taking place.

Note

*Regular expressions are a system for concisely describing classes of strings. If you aren't familiar with regular expressions, you can get by with the following short summary: "foo" would match any file whose name contains the string "foo" and "foo.*bar" would match any file whose name contains "foo", followed by any number of characters, followed by the string "bar." That's because normal substrings match themselves, but "." and "*" are special. "." matches any character and "*" means match any number of the preceding character, including zero. The "^" and "$" signs mean match at the beginning and end of the string, respectively. Thus, "^foo.*bar.*baz$" would match any string beginning with "foo", containing "bar" somewhere in the middle, and ending with "baz." That's all we'll go into here; this summary is a very abbreviated subset of full regular expression syntax. Regular expressions require detailed study to be used intelligently and effectively. The reader is well advised to read one of the good books on this subject.*

The commitinfo file is for generic hooks you want run on every commit. Here are some example commitinfo lines:

```
^a-subdir*      /usr/local/bin/check-asubdir.sh
ou              /usr/local/bin/validate-project.pl
```

So any commit into myproj/a-subdir/ would match the first line, which would then run the check-asubdir.sh script. A commit in any project whose name (actual repository directory name, not necessarily module name) contained the string "ou" would run the validate-project.pl script, *unless* the commit had already matched the previous a-subdir line.

In place of a regular expression, the word **DEFAULT** or **ALL** can be used. The **DEFAULT** line (or the first **DEFAULT** line, if there are more than one) will be run if no regular expression matches, and each of the **ALL** lines will be run in addition to any other lines that may match.

Note

The file names passed to the program do not refer to RCS files—they point to normal files, whose contents are exactly the same as the working-copy files being committed. The only unusual aspect is that CVS has them temporarily placed inside the repository, so they'll be available to programs running on the machine where the repository is located.

The loginfo file is similar to commitinfo, except that instead of acting on the files' contents, it acts on the log message. The left side of the loginfo file contains regular expressions, including possibly **DEFAULT** and **ALL** lines. The program invoked on the right side receives the log message on its standard input; it can do whatever it wants with that input.

The program on the right side can also take an arbitrary number of command-line arguments. One of those arguments can be a special "%" code, to be expanded by CVS at runtime, as follows:

```
%s    ------>      name(s) of the file(s) being committed
%V    ------>      revision number(s) before the commit
%v    ------>      revision number(s) after the commit
```

The expansion always begins with the path to the repository (done for backward-compatibility), followed by the per-file information. For example, if the files committed were foo, bar, and baz, then **%s** would expand into

```
myproj  foo  bar  baz
```

whereas **%V** would expand to show their old revision numbers

```
myproj  1.7  1.134  1.12
```

and **%v** their new revision numbers:

```
myproj  1.8  1.135  1.13
```

There can only be one "%" expression per line in the loginfo file. If you want to use more than one of the codes, you must enclose them in curly braces after the "%" sign—this will expand them into a series of comma-separated sublists, each containing the corresponding information for one file in the commit. For instance, **%{sv}** would expand to

```
myproj  foo,1.8  bar,1.135  baz,1.13
```

whereas **%{sVv}** would expand to:

```
myproj  foo,1.7,1.8  bar,1.134,1.135  baz,1.12,1.13
```

(You might have to look carefully to distinguish the commas from the periods in those examples.)

Here is a sample loginfo file:

```
^myproj$   /usr/local/newrepos/CVSROOT/log.pl -m myproj-devel@foobar.com %s
ou         /usr/local/bin/ou-notify.pl  %{sv}
DEFAULT    /usr/local/bin/default-notify.pl  %{sVv}
```

In the first line, any commit in the myproj subdirectory of the repository invokes "log.pl," passing it an email address (to which log.pl will send an email containing the log message), followed by the repository, followed by all the files in the commit.

In the second line, any commit in a repository subdirectory containing the string "ou" will invoke the (imaginary) "ou-notify.pl" script, passing it the repository followed by the file names and new revision numbers of the files in the commit.

The third line invokes the (equally imaginary) default-notify.pl script for any commit that didn't match either of the two previous lines, passing it all possible information (path to repository, file names, old revisions, and new revisions).

The verifymsg and rcsinfo Files

Sometimes you might just want a program to automatically verify that the log message conforms to a certain standard and to stop the commit if that standard is not met. This can be accomplished by using verifymsg, possibly with some help from rcsinfo.

The verifymsg file is the usual combination of regular expressions and programs. The program receives the log message on standard input; presumably it runs some checks to verify that the log message meets certain criteria, then it exits with status zero or nonzero. If the latter, the commit just fails.

Commit Emails

The loginfo file is how one sets up commit emails—automated emails that go out to everyone working on a project whenever a commit takes place. (It might seem counterintuitive that this is done in loginfo instead of commitinfo, but the point is that one wants to include the log message in the email.) The program to do the mailing—contrib/log.pl in the CVS source distribution—can be installed anywhere on your system. It is customary to put it in the repository's CVSROOT/ subdirectory, but that's just a matter of taste.

You might need to edit log.pl a bit to get it to work on your system, possibly changing the first line to point to your Perl interpreter, and maybe changing this line

```
$mailcmd = "| Mail -s 'CVS update: $modulepath'";
```

to invoke your preferred mailer, which might or might not be named "Mail." Once you've got it set the way you like it, you can put lines similar to these into your loginfo:

```
listerizer CVSROOT/log.pl %s -f CVSROOT/commitlog -m listerizer@red-bean.com
RoadMail   CVSROOT/log.pl %s -f CVSROOT/commitlog -m roadmail@red-bean.com
bk/*score  CVSROOT/log.pl %s -f CVSROOT/commitlog -m bkscore-devel@red-bean.com
```

The **%s** expands to the names of the files being committed; the **-f** option to log.pl takes a file name, to which the log message will be appended (so CVSROOT/commitlog is an ever-growing file of log messages). The **-m** flag takes an email address, to which log.pl will send a message about the commit. The address is usually a mailing list, but you can specify the **-m** option as many times as necessary in one log.pl command line.

Meanwhile, the left side of rcsinfo has the usual regular expressions, but the right side points to template files instead of programs. A template file might be something like this

```
Condition:
Fix:
Comments:
```

or some other collection of fields that a developer is supposed to fill out to create a valid log message. The template is not very useful if everyone commits using the **-m** option explicitly, but many developers prefer not to do that. Instead, they run

```
yarkon$ cvs commit
```

and wait for CVS to automatically fire up a text editor (as specified in the **EDITOR** environment variable). There they write a log message, then save the file and exit the editor, after which CVS continues with the commit.

In that scenario, an rcsinfo template would insert itself into the editor before the user starts typing, so the fields would be displayed along with a reminder to fill them in. Then when the user commits, the appropriate program in verifymsg is invoked. Presumably, it will check that the message does follow that format, and its exit status will reflect the results of its inquiry (with zero meaning success).

As an aid to the verification programs, the path to the template from the rcsinfo file is appended as the last argument to the program command line in verifymsg; that way, the program can base its verification process on the template itself, if desired.

Note that when someone checks out a working copy to a remote machine, the appropriate rcsinfo template file is sent to the client as well (it's stored in the CVS/ subdirectory of the working copy). However, this means that if the rcsinfo file on the server is changed after that, the client won't see the changes without re-checking out the project (merely doing an update won't work).

Note also that in the verifymsg file, the **ALL** keyword is not supported (although **DEFAULT** still is). This is to make it easier to override default verification scripts with subdirectory-specific ones.

The taginfo File

What loginfo does for log messages, taginfo does for tags. The left side of taginfo is regular expressions, as usual, and the right side is programs. Each program is automatically handed arguments when CVS **tag** is invoked, in this order:

```
arg 1:          tag name
arg 2:          operation ("add" => tag, "mov" => tag -F, "del" => tag -d)
arg 3:          repository
arg 4, 5, etc:  file revision [file revision ...]
```

If the program returns nonzero, the tag is aborted.

We haven't covered the **-F** option to **tag** before now, but it's exactly what the above implies: a way to move a tag from one revision to another. For example, if the tag "Known_Working" is attached to revision 1.7 of a file and you want it attached to revision 1.11 instead, do this

```
cvs tag -r 1.11 -F Known_Working foo.c
```

which removes the tag from 1.7, or wherever it was previously in that file, and puts it at 1.11.

The cvswrappers File

The redundantly named cvswrappers file gives you a way to specify that certain files should be treated as binary, based on their file name. CVS does not assume that all files with a .jpg extension are JPG image data, for example, so it doesn't automatically use **-kb** when adding JPG files. Nonetheless, for certain projects, it would be very useful to be able to simply designate all JPG files as binary. Here is a line in cvswrappers to do that:

```
*.jpg -k 'b'
```

The **'b'** is separate and in quotes because it's not the only possible RCS keyword expansion mode; one could also specify **'o'**, which means not to expand $ keywords but to do newline conversion. However, **'b'** is the most common parameter.

There are a few other modes that can be specified from the wrappers file, but they're for such rare situations that they're probably not worth documenting here. See the node "Wrappers" in the Cederqvist if you're curious.

The editinfo File

This file is obsolete, even though it's still included in distributions. Just ignore it.

The notify File

This file is used in conjunction with CVS's "watch" features, which are described in Chapter 4. Nothing about it will make sense until you understand what watches are (they're a useful but nonessential feature), so see Chapter 4 for details about this file and about watches.

The checkoutlist File

If you look inside CVSROOT/, you'll see that working copies of the files exist side by side with their RCS revision files:

```
yarkon$ ls /usr/local/newrepos/CVSROOT
checkoutlist     config,v       history     notify     taginfo
checkoutlist,v   cvswrappers    loginfo     notify,v   taginfo,v
commitinfo       cvswrappers,v  loginfo,v   passwd     verifymsg
```

```
commitinfo,v     editinfo       modules     rcsinfo    verifymsg,v
config           editinfo,v     modules,v   rcsinfo,v

yarkon$
```

CVS pays attention only to the working versions, not the RCS files, when it's looking for guidance on how to behave. Therefore, whenever you run **commit** on your working copy of CVSROOT/ (which might even, after all, be checked out to a different machine), CVS automatically updates any changed files in the repository itself. You will know that this has happened because CVS will print a message at the end of such commits:

```
yarkon$ cvs ci -m "added mp and asub modules" modules
Checking in modules;
/usr/local/newrepos/CVSROOT/modules,v  <--  modules
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database
```

CVS automatically knows about the standard administrative files and will rebuild them in CVSROOT/ as necessary. If you decide to put custom files in CVSROOT/ (such as programs or rcsinfo template files), you'll have to tell CVS explicitly to treat them the same way.

That's the purpose of the checkoutlist file. It has a different format from most of the files we've looked at so far:

```
FILENAME     ERROR_MESSAGE_IF_FILE_CANNOT_BE_CHECKED_OUT
```

Here are some examples:

```
log.pl           unable to check out / update log.pl in CVSROOT

bugfix.tmpl      unable to check out / update bugfix.tmpl in CVSROOT
```

Certain files in CVSROOT are traditionally not kept under revision control. One such file is the history file, which keeps a running record of all actions in the repository, for use by the **cvs history** command (which lists checkout, update, and tag activity for a given file or project directory). Incidentally, if you just remove the history file, CVS will obligingly stop keeping that log.

Tip

Sometimes the history file is the cause of permission problems, and the easiest way to solve them is to either make the file world-writable or just remove it. You must, however, resist this temptation, because making it world-writable introduces a host of security problems. Instead, try to understand what the exact issue is with the permissions of the history file and extend those permissions cautiously.

Another "unrevisioned" administrative file is passwd, the assumption being that having it checked out over the network might compromise the passwords (even though they're encrypted). You'll have to decide based on your own security situation whether you want to add passwd to checkoutlist or not; by default, it is not in checkoutlist.

Two final notes about the CVSROOT/ directory: It is possible, if you make a big enough mistake, to commit an administrative file that is broken in such a way as to prevent any commits from happening at all. If you do that, naturally you won't be able to commit a fixed version of the administrative file! The solution is to go in and hand-edit the repository's working copy of the administrative file to correct the problem; the whole repository might stay inaccessible until you do that.

Also, for security's sake, make sure your CVSROOT/ directory is writable only by users you trust (by "trust," we mean you trust both their intentions and their ability not to compromise their password). The *info files give people the ability to invoke arbitrary programs, so anyone who can commit or edit files in the CVSROOT/ directory can essentially run any command on the system. That's something you should always keep in mind.

Finding Out More

Although this chapter tries to give a complete introduction to installing and administering CVS, we have left out things that are either too rarely used to be worth mentioning or already well documented in the Cederqvist manual. The latter category includes setting up the other remote access methods: **rsh/ssh**, kserver (Kerberos 4), and GSSAPI, or Generic Security Services API (which includes Kerberos 5, among other things). It should be noted that nothing special needs to be done for **rsh/ssh** connections, other than making sure that the user in question can log in to the repository machine using **rsh** or **ssh**. If the user can and CVS is installed on both client and server, and the user has the right permissions to use the repository directly from the server machine, then he or she should be able to access the repository remotely via the **:ext:** method.

We have deferred descriptions of certain specialized features of CVS to later chapters, so they can be introduced in contexts where their usefulness is obvious. General CVS troubleshooting tips are found in Chapter 5.

Although it's not necessary to read the entire Cederqvist manual, you should familiarize yourself with it; it is an invaluable reference tool. If for some reason you don't have Info working on your machine and don't want to print the manual, you can browse it online at **http://durak.org/cvswebsites/doc/** or **www.loria.fr/~molli/cvs/doc/cvs_toc.html**.