

3

Software Configuration Management

A software configuration management (SCM) system consists of tools and processes used to manage the source code of a project and to assess its state. With these tools, management and developers can do the following:

- Peruse the source code managed by the SCM system
- Obtain a local copy of the source code
- Submit modifications made to the source code by developers
- Understand which changes have been made to the source code by developers
- Determine the state of the source code (for example, whether it compiles)
- Log and track defects associated with binaries created from the source code

In a cross-platform project, certain aspects of an SCM system take on added importance. In this chapter, I describe the tools that were most important to Netscape and Mozilla in this regard:

- CVS, the Concurrent Version System, which was used to maintain and manage a central copy of the Netscape/Mozilla source code
- Bugzilla, used to track defects in programs generated from the source code stored in CVS
- Tinderbox, used to determine the state of the source code as changes were made to the copy stored in CVS
- Patch, a tool that fosters the distribution and communication of changes among developers (in particular, between a developer and code reviewers) prior to it being permanently stored in CVS

These tools, combined with some best practices for using them that I describe in this chapter, helped greatly in ensuring that Netscape shipped cross-platform products that were similar in features and overall quality, and did so on the same schedule.

Item 11: Use a Cross-Platform Bug Reporting and Tracking System

A major component of a cross-platform toolset is the bug reporting and tracking system (which I refer to as a “bug system” here for brevity). A bug system is used by developers and testers to report defects and issues encountered during the software development and testing phases of a project. In general, a bug system should allow the reporter to specify the problem encountered, identify the context or state of the system at the time the bug was discovered, and list the steps required to reproduce the bug. The bug system also must allow for the tracking of any issues it contains. In terms of tracking, bugs generally go through the following states:

- **New.** A bug that has been discovered, but has not been investigated by a developer.
- **Assigned.** The developer acknowledges the bug and is investigating.
- **Resolved.** The developer has fixed the bug, or has some other resolution (for example, unable to reproduce the problem, or the feature is working as designed).
- **Verified.** QA/test has acknowledged the resolution of the developer, and has verified it to be correct.

In addition, a bug can be in the reopened state, an indication that a once-verified bug has resurfaced in testing after originally being verified by QA.

Finally, a bug system should allow someone to specify the relative priority of bugs filed against a product. Initially, this will be the person filing the bug, but ideally, it is done in a “triage” session, with participation of the following teams: development, QA/test, and product management. Tracking priorities is not easy, it takes discipline for people to sit down for an hour or two and grind through a list of bugs, but doing so allows developers to prioritize their work, and allows those who are making ship/no-ship decisions the opportunity to objectively decide what state the project is in.

When it comes to cross-platform development, you should always look for a couple of attributes and features when selecting a bug system for use in your project.

Accessibility

Perhaps the most important cross-platform attribute you should look for when selecting a bug system is accessibility. In a cross-platform development project, everyone will have his or her preferred development platform (see Item 1). For some developers, it will be Mac OS X, and for many it will be Windows. If the bug system is not accessible to everyone on the team via their native platforms, it risks not being used by those who would rather not boot into one of the operating systems that the bug system does support. Therefore, the bug system itself needs to be cross-platform. And, it must support all of your tier-1 platforms.

These days, by and large, the accessibility requirement is best met by using a Web-based bug system. Web browsers are available for every platform, including PDAs and mobile phones. Bug reporting/tracking systems based on LAMP (Linux, Apache, MySQL, PHP) or Java/JSP are available.

Ability to Track Platform-Specific Bugs

In some cases, a bug will be reproducible on all platforms, and in other cases, only a subset of the supported platforms will exhibit the given problem. It is critical that the platforms affected by the problem be tracked. The bug system should allow the person filing the bug, or even someone updating the bug at a later time, to specify the platform(s) affected by any given bug. Doing so will allow someone searching the bug database to specify a subset of the supported platforms as search criteria (for example, “find all assigned Linux bugs”). By specifying the platforms as part of the search criteria, platform specialists will be able to quickly identify bugs that affect their platform, and management will be able to determine the number of open (and resolved) issues on a per-platform basis.

Bugzilla

Most of my experience with bug reporting systems has been with a bug tracking system named Bugzilla—yet another result of the Netscape/Mozilla project. Bugzilla is implemented as a Web application, allowing the database to be viewed from any platform that supports a Web browser. Bugzilla has

been in active development for years, and is used by hundreds of organizations. Because Bugzilla originated during the development of the cross-platform Netscape/Mozilla system (it replaced a Netscape-internal bug system in use during the development of the 4.x browser called bugsplat), it has always supported cross-platform development directly.

Installing Bugzilla is pretty straightforward, and is well documented. A basic installation of Bugzilla can be done in about an hour, depending on your setup. (That was my experience on a stock Red Hat Fedora Core 4 system, and your results may vary.)

The following instructions are based on a Fedora Core 4 setup, using Apache and MySQL. (The default versions of Apache, Perl, and MySQL that come with FC4 are suitable for use with Bugzilla 2.22.)

Visit www.bugzilla.org, and download the latest stable release (2.22 as of this writing) onto the Linux machine that you will use to host the Bugzilla server. Then:

```
$ su
# tar -zxvf bugzilla-2.22.tar.gz
# mkdir /var/www/html/bugzilla
# cd bugzilla-2.22
# cp -r * /var/www/html/bugzilla
# cd /var/www/html/
# chown -R apache bugzilla
# chgrp -R apache bugzilla
```

The next step is to install required Perl modules. To do this, enter the following:

```
# perl -MCPAN -e 'install "Bundle::Bugzilla"'
```

You may be prompted for inputs. If so, just take whatever default responses are offered. To validate that the required modules are installed, issue the following (as root):

```
# cd /var/www/html/bugzilla
# perl checksetup --check-modules
```

When that completes, you need to rerun `checksetup` once again:

```
# perl checksetup.pl
```

This will create a file named `localconfig`. To configure Bugzilla to work with MySQL, follow the instructions at www.bugzilla.org/docs/2.22/html/configuration.html#mysql. Finally, rerun `checksetup.pl` one last time. The `checksetup.pl` script will create the Bugzilla database and prompt you for information needed to set up an administrator account in the Bugzilla database. Following this, you should be able to use the following URL to access bugzilla from your local Web browser:

`http://localhost/bugzilla`

If you have any problems with these instructions (or have a setup that differs from mine), refer to `file://var/www/html/bugzilla/docs/index.html` for help.

After you have installed Bugzilla, you need to add products to the bug database, and modify the list of platforms (hardware and operating systems) that can be specified when bugs are filed in the database. To do this, log in as the administrator; you should see a screen similar to the one shown in Figure 3-1.

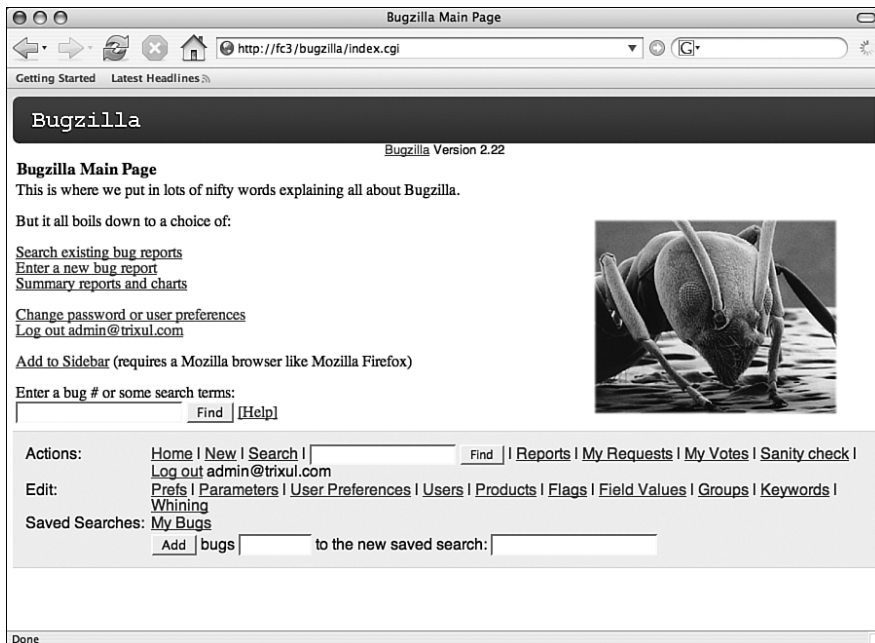


Figure 3-1 Bugzilla main screen

In the bottom half of Figure 3-1, there is a link labeled Products. Clicking that link takes you to a screen on which you can enter the name and attributes of a product. Enter the product name and a description, accept defaults for the remaining fields, and click the Add button. You should get something like Figure 3-2.

Select product

http://fc3/bugzilla/editproducts.cgi

Getting Started Latest Headlines

Bugzilla

Bugzilla Version 2.22

Select product

Edit product...	Description	Open For New Bugs	Votes Per User	Maximum Votes Per Bug	Votes To Confirm	Action
Trixul	The Trixul cross-platform GUI toolkit	Yes	0	10000	0	Delete

[Redisplay table with bug counts \(slower\)](#)

Add a product.

Actions: [Home](#) | [New](#) | [Search](#) | [Find](#) | [Reports](#) | [My Requests](#) | [My Votes](#) | [Sanity check](#) | [Log out admin@trixul.com](#)

Edit: [Prefs](#) | [Parameters](#) | [User Preferences](#) | [Users](#) | [Products](#) | [Flags](#) | [Field Values](#) | [Groups](#) | [Keywords](#) | [Whining](#)

Saved Searches: [My Bugs](#)

[Add](#) bugs to the new saved search:

Done

Figure 3-2 Products screen

The next step manages the list of supported operating systems. To view the list, click the Field Values link (visible in Figures 3-1 and 3-2). Then, click the OS link in the resulting page, and you should see the list of supported operating systems (All, Windows, Mac OS, Linux, and Other, as in Figure 3-3). By clicking the operating system names to edit them, change the value Windows to Windows XP, the value Mac OS to be Mac OS X 10.4, and the value Linux to be Fedora Core 4. I personally think it is better to be fine-grained when it comes to the list of operating systems; Linux, for example, is far too vague to be truly useful. You can also add or remove operating systems using the provided controls.

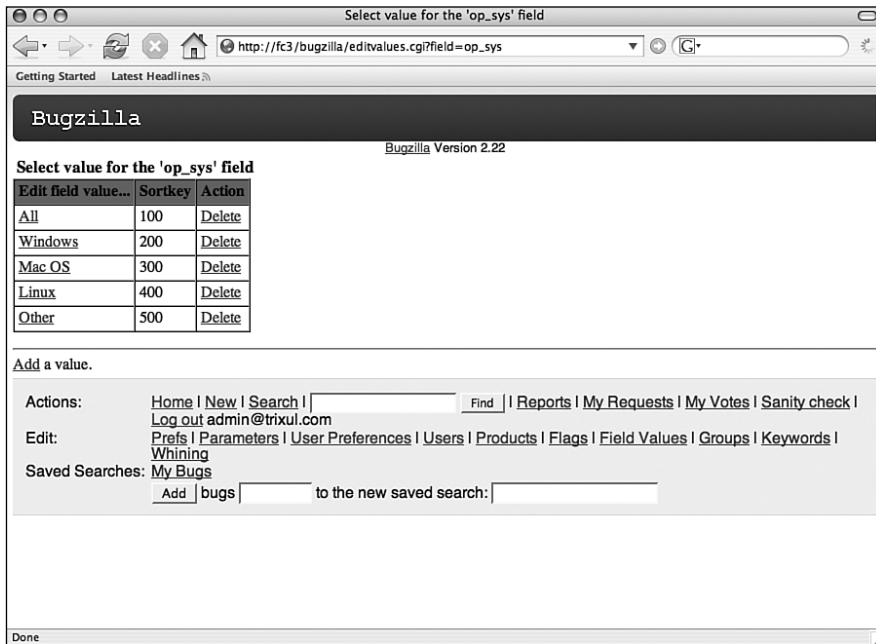


Figure 3-3 Platforms screen

To modify the list of supported hardware, click the Field Values link again, click the Hardware link on the resulting page, and then modify the list of supported hardware types so that it is consistent with your supported hardware platforms. Again, being fine-grained can help in diagnosing problems. For example, instead of Macintosh, you might add choices based on type and CPU (for example, PowerBook and MacBook).

Before any bug can be filed against a product, components and version numbers need to be added for the product. This can be done by clicking the Product link, clicking the product name in the resulting table, and then clicking the Edit Components and Edit Versions links, respectively. Components are specific areas of a product. For Trixul, I defined three components: JavaScript Integration, Layout engine, and liblayout. Additional components can be added by the administrator as needed. Version numbers can be any text string that helps to identify the versions of the product; I specified 0.1.0 and 1.0. Figure 3-4 illustrates the resulting product summary page displayed by Bugzilla. As soon as at least one component and one version number has been specified, users should be able to add new bugs to the database.

Edit Product

Product:

Description:

Closed for bug entry: ☐

Maximum votes per person:

Maximum votes a person can put on a single bug:

Number of votes a bug in this product needs to automatically get out of the UNCONFIRMED state:

Edit components: JavaScript Integration: Bugs related to JavaScript integration
Layout engine: Bugs related to core, abstract layout engine
liblayout: Bugs related to general architecture of the trixul layout library

Edit versions: 0.1.0
1.0

Edit Group Access Controls: no groups

Bugs: 0

Update

Figure 3-4 Product Summary Page

Returning to the cross-platform issues of platform and hardware, let's see how they are specified by filing a bug against Trixul. One problem with Trixul is that under Microsoft Windows XP, the layout engine is built as an executable, whereas on all other platforms, the layout engine is built as a shared library that is then linked to a small executable. The reason for this bug (I believe) is a difficulty with linking a mixed (native and Common Language Infrastructure [CLI]) .NET application to a shared library using Visual Studio .NET 2003.

To file this bug, I logged on to Bugzilla, clicked the Enter a New Bug Report link, filled out the form as shown in Figure 3-5, and then clicked Commit.

Notice how I selected Windows XP in the OS field, and PC (Intel Pentium 4) in the Platform field.

Platform-specific bugs can be easily located using the search feature of Bugzilla. To do so, visit the main Bugzilla page and click the Search Existing Bug Reports link. The resulting page allows you to specify a wide variety of search criteria, including the OS and Platform fields. Figure 3-6 illustrates a search of all open bugs filed against both Fedora Core 4 and Windows XP on the Pentium 4 platform.

Enter Bug: Trixul

http://fc3/bugzilla/enter_bug.cgi

Getting Started Latest Headlines

Bugzilla

Bugzilla Version 2.22

Enter Bug: Trixul

Before reporting a bug, please read the [bug writing guidelines](#), please look at the list of [most frequently reported bugs](#), and please [search](#) for the bug.

Reporter: slogan621@gmail.com

Version: 0.1.0

Product: Trixul

Component: JavaScript Integration

Platform: PC (Intel Pentium 4)

OS: Windows XP

Priority: P1

Severity: blocker

Initial State: NEW

Assign To: slogan621@gmail.com

Cc:

URL: http://

Summary: liblayout not supported on Windows XP

Description: On Windows XP, the layout engine is created as an executable, not as a library (as is done for MacOS X and Linux). A plausible assumption as to the cause of the problem is that Visual Studio .NET has bugs related to mixed binaries (CLR and Native), and that an upgrade to the latest version of Visual Studio will fix the problem. An attempt was made to do the work under .NET 2003; with the 1.1 .NET framework. I also tried using .NET 2.0 but had problems with that as well. A patch with all the makefile changes will be attached to this bug

Done

Figure 3-5 Filing a new bug report

Search for bugs

http://fc3/bugzilla/query.cgi

Getting Started Latest Headlines

Bugzilla

Bugzilla Version 2.22

Find a Specific Bug

Advanced Search

Give me some help (reloads page.)

Summary: contains all of the words/strings

Search

Product: Trixul

Component: JavaScript Integration

Version: 0.1.0

A Comment: contains the string

The URL: contains all of the words/strings

Status: UNCONFIRMED

Resolution: FIXED

Severity: blocker

Priority: P1

Hardware: All

OS: All

Email and Numbering

Bug Changes

Done

Figure 3-6 Searching for bugs

If you like, you can copy and paste the URL of the search result page into a Web page or e-mail, and use that as a quick way to find all bugs related to a specific platform/operating system combination. When using Bugzilla I generally make a Web page consisting of links to all queries that I consider essential for the project I am working on; clicking a link is much easier than revisiting the Bugzilla search page and reentering all the parameters of the desired query each time I go to look at my list of bugs.

That's pretty much it for Bugzilla. Using the general guidelines I just provided, it is fairly easy to set up and maintain a database that is capable of helping you ensure that platform-specific issues are tracked accurately.

Item 12: Set Up a Tinderbox

Tinderbox is a tool that was developed initially at Netscape, but that is now open source software maintained by the Mozilla project. Tinderbox is designed to manage the complexity one encounters when developing software, especially in terms of large-scale cross-platform software that involves a widely distributed team of developers. Tinderbox is particularly useful in cross-platform projects, as you will see. Coupled with a system known as bonsai, the goals of Tinderbox are fairly simple:

- Communicate any and all changes made over time to the source code repository to the entire development team, in a centralized location, as soon as the changes have been made.
- Communicate the overall health of the repository by continually pulling and building the source code on each supported platform. For each pull and build cycle, a pass/fail status is reported to a centralized location. This allows developers to determine when they should update their local trees to avoid pulling source code that will not build (or run) correctly.
- Combining the above, Tinderbox can be used to assign accountability of the health of the tree to specific individuals and/or changes to the repository. Knowing this information helps get problems solved as quickly and accurately as possible.

Basically, Tinderbox is a group of machines that continually pull and build the contents of a CVS repository (see Item 13), and a server that retrieves and reports the status of these builds on a Web server that everyone in the organization can monitor. Tinderbox is currently supported as three versions. Version 1, perhaps the most widely used, was developed by Netscape/Mozilla, and is still in use by mozilla.org. Tinderbox 2.0 is a

rewrite of version 1, providing essentially the same feature set. The goal of Tinderbox 2.0 was to essentially clean up the implementation of version 1. Both Tinderbox 1 and Tinderbox 2 are available from mozilla.org. Tinderbox 3 is a more recent version, available as a tarball from John Keiser, an ex-Netscape developer. Tinderbox 3 adds a number of desirable features, and strives to make Tinderbox easier to set up and administer.

Figure 3-7 illustrates the Web page displayed by a Tinderbox server. (You can access a large number of live Tinderboxen by visiting <http://Tinderbox.mozilla.org>.)

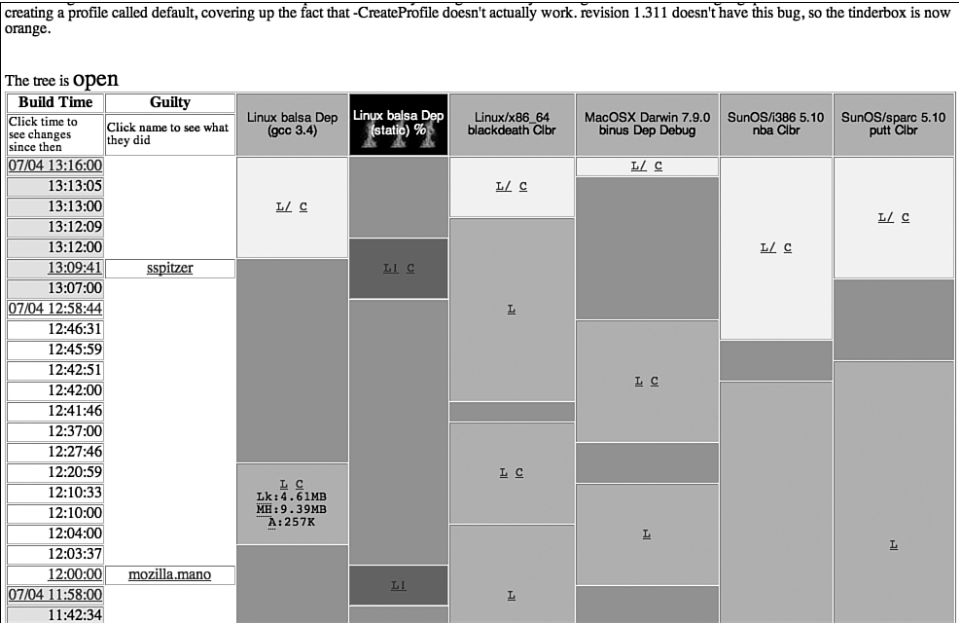


Figure 3-7 Tinderbox

The Tinderbox in Figure 3-7 illustrates the state of Mozilla’s Seamonkey reporting the health of some of the “port” platforms that are defined by the Mozilla project. (Seamonkey was the code name used by Netscape/Mozilla during the development of the Mozilla browser suite. A port is a platform that is not considered to be tier-1 by Mozilla.)

The use of Tinderbox is pervasive in the software development community. Not only is it used by mozilla.org, but by other open source projects (for example, OSDL) and in commercial development (AOL, for example). Tinderbox is particularly well-suited to cross-platform development, as we discuss later.

The Tinderbox Web page consists of a table viewed as a series of mutually exclusive columns that are organized from left to right. In the first column (Build Time), each row contains a timestamp that can be used to identify the time associated with events that are represented in the remaining columns of the table. The second column identifies each check-in made by developers; the time of these check-ins can easily be determined by looking at the corresponding row in the Build Time column. The remaining columns each represent a specific platform that is being reported on by the Tinderbox. (An organization may have several Tinderboxes, each reporting a specific group of builds. You can see an example of this by visiting <http://tinderbox.mozilla.org/showbuilds.cgi>.)

Any given column represents a build machine, and a platform, and contains a series of colored boxes. Green boxes indicate a successful build of the repository for that platform on that machine. Conversely, a red box indicates a failed build, and a yellow box indicates a build that is currently in the process of being produced. Furthermore, the lower edge of any of these boxes represents the start of a pull and build cycle, and the upper edge represents the time of completion. The time corresponding to both of these events, for a particular colored box, can be determined by looking at the timestamp at the same row in column one as the upper or lower edge of that box. For example, the uppermost failed build in Figure 3-7 (Linux balsa Dep (static), represented by column four of the table) was started at about 13:07, and failed about five minutes later, at about 13:12.

Let's take a closer look at this failed build, and see what we can infer about it. It is clear that the check-in by sspitzer at 13:09:41 did not result in the failure of the build for two reasons. First of all, the Linux balsa Dep build was already burning prior to sspitzer's check-in. (See the red box in the same column that completed around 12:00, and also notice how the lower edge of the uppermost red box is lower than the entry for sspitzer's check-in.) Another piece of evidence that sspitzer is not the cause of the problem is that each of the other Linux platform builds are green. (Generally, one finds that Linux builds of Mozilla are generally are either all red at the same time, or all green.) Finally, and perhaps most important, we can see that the build was previously red at noon (12:00), and had not gone through a green cycle since then. (Gray portions of the Tinderbox indicate no build was in progress, or the progress of a build was not reported to the Tinderbox server.)

For largely the same reasons cited previously for sspitzer, we can also infer that mozilla.mano is not to blame for the redness of the build, either.

The tree was already red prior to his or her check-in, and the other Linux builds were not affected.

Let's say I am not entirely sure that sspitzer is not to blame, and want to take a closer look at what the exact cause of the broken build might be. There are several other facilities provided by Tinderbox that you can use to drill down for further information. The L (or L1) link inside of the red box can be used to obtain a log of the build; this log contains compiler and perhaps linker output that should identify what caused the build to break. Clicking the L1 link gives the result shown in Figure 3-8.

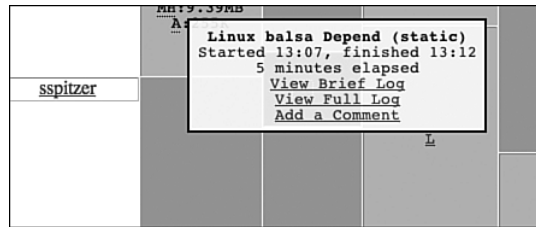


Figure 3-8 L1 link output

Clicking View Brief Log results in Figure 3-9, which indicates a problem building libmozjs.so, the shared library that contains the implementation of the Mozilla Spidermonkey JavaScript engine (which is used in Trixul; see Chapter 9, “Developing a Cross-Platform GUI Toolkit in C++”).

```

Build Error Summary

gmake[5]: *** [libmozjs.so] Error 1
gmake[5]: Leaving directory `/builds/tinderbox/SeaMonkey/Linux_2.4.7-10_Depend/mozilla/js/src'
gmake[4]: *** [libs] Error 2
gmake[4]: Leaving directory `/builds/tinderbox/SeaMonkey/Linux_2.4.7-10_Depend/mozilla/js'
gmake[3]: *** [libs_tier_2] Error 2
gmake[3]: Leaving directory `/builds/tinderbox/SeaMonkey/Linux_2.4.7-10_Depend/mozilla'
gmake[2]: *** [tier 2] Error 2
gmake[2]: Leaving directory `/builds/tinderbox/SeaMonkey/Linux_2.4.7-10_Depend/mozilla'
gmake[1]: *** [alldep] Error 2
gmake[1]: Leaving directory `/builds/tinderbox/SeaMonkey/Linux_2.4.7-10_Depend/mozilla'
gmake: *** [alldep] Error 2

Build Error Log

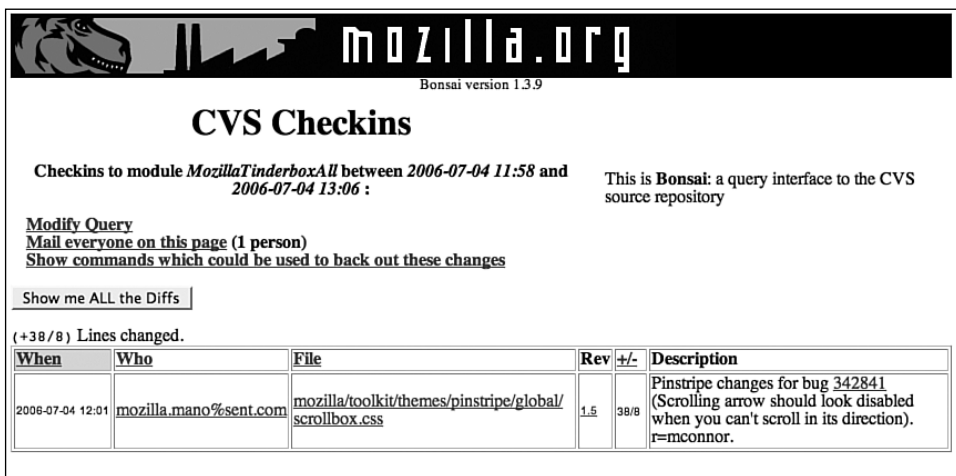
Skipping 705 Lines...

/lib/libc.so.6: undefined reference to `_dl_lookup_symbol@GLIBC 2.0'
/lib/libc.so.6: undefined reference to `_libc_stack_end@GLIBC 2.1'
/lib/libc.so.6: undefined reference to `_dl_argv@GLIBC 2.2'
/usr/lib/gcc-lib/i386-redhat-linux/2.96/../../../../libpthread.so: undefined reference to `_dl_cputlock_offset'
/lib/libc.so.6: undefined reference to `_dl_loaded@GLIBC 2.1'
/lib/libc.so.6: undefined reference to `_dl_origin_path@GLIBC 2.1.1'
/lib/libc.so.6: undefined reference to `_dl_check_map_versions@GLIBC 2.2'
/lib/libc.so.6: undefined reference to `_dl_map_object@GLIBC 2.0'
/lib/libc.so.6: undefined reference to `_dl_main_searchlist@GLIBC 2.1'

```

Figure 3-9 View brief log output

The other major source of input that I would want to consider is an understanding of what portion of the tree was impacted by sspitzer. If the check-in he made does not correlate to the error messages displayed in the log (Figure 3-9), I can eliminate him from the “blame” list and focus my search elsewhere. I can do this in two ways. First, by clicking the C link in the red box; this will display a list of checkins that were made prior to the start of the build. It is these checkins that likely would be the cause of any state change in the build (that is, going from red to green, or from green to red, which is not the case here since the tree was already red at the 12:00 hour). Figure 3-10 shows the result of clicking the C link.



mozilla.org
Bonsai version 1.3.9

CVS Checkins

Checkins to module *MozillaTinderboxAll* between *2006-07-04 11:58* and *2006-07-04 13:06* :

This is **Bonsai**: a query interface to the CVS source repository

[Modify Query](#)
[Mail everyone on this page \(1 person\)](#)
[Show commands which could be used to back out these changes](#)

(+38/8) Lines changed.

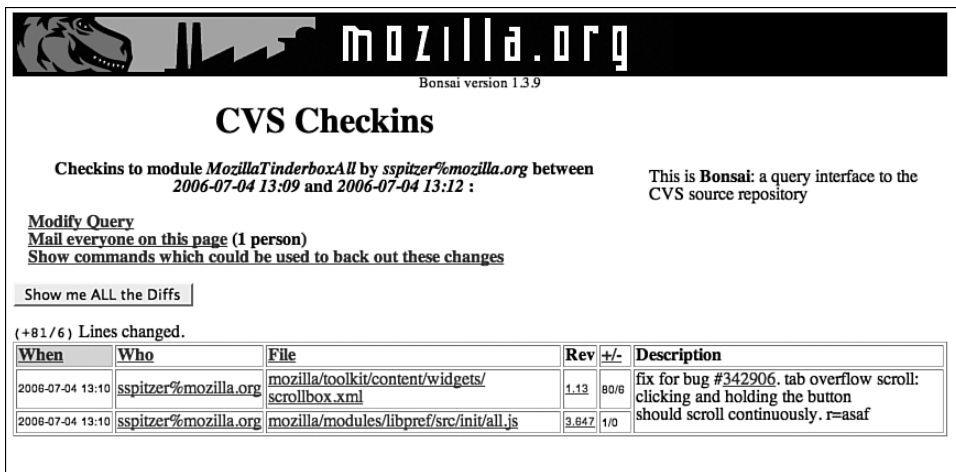
When	Who	File	Rev	+/-	Description
2006-07-04 12:01	mozilla.mano%sent.com	mozilla/toolkit/themes/pinstripe/global/scrollbox.css	1.5	38/8	Pinstripe changes for bug 342841 (Scrolling arrow should look disabled when you can't scroll in its direction). r=mconnor.

Figure 3-10 List of checkins

This result clearly confirms that sspitzer is not on the blame list—and it also shows that the problem is not related to a check-in by mozilla.mano, because this change was made to a Cascading Style Sheets (CSS) source file that would have no impact on the JavaScript engine.

The final technique to eliminate sspitzer from the blame list (short of sending him an e-mail and asking him whether his check-in caused the build failure) would be to click the sspitzer link in column one and see what changes he made. Doing so gives us Figure 3-11.

Once again, these are changes to Extensible Markup Language (XML) markup and JavaScript sources that have no bearing whatsoever on the stability of the JavaScript engine, the failure of which was clearly the cause of the red tree.



CVS Checkins

Checkins to module *MozillaTinderbox:All* by *sspitzer%mozilla.org* between
2006-07-04 13:09 and 2006-07-04 13:12 :

This is Bonsai: a query interface to the
CVS source repository

[Modify Query](#)
[Mail everyone on this page \(1 person\)](#)
[Show commands which could be used to back out these changes](#)

[Show me ALL the Diffs](#)

(+81/6) Lines changed.

When	Who	File	Rev	+/-	Description
2006-07-04 13:10	sspitzer%mozilla.org	mozilla/toolkit/content/widgets/ scrollbox.xml	1.13	80/6	fix for bug #342906. tab overflow scroll: clicking and holding the button should scroll continuously. r=asaf
2006-07-04 13:10	sspitzer%mozilla.org	mozilla/modules/libpref/src/init/all.js	3.647	1/0	

Figure 3-11 List of checkins made by sspitzer

In terms of cross-platform development, it is best not to rely on a Tinderbox to find all of your portability problems. At Netscape, developers were required to ensure that any changes they were considering for checkin built and executed cleanly on each tier-1 platform, before submitting the changes to the repository (see Item 4). However, you can never be sure that all developers will follow this rule all the time, without exception. Because of this, having a Tinderbox monitor the health of the repository is an especially good idea. It isolates the problems in terms of change made, time of the change, and developer who made the change, focusing the area of investigation to a small fraction of the overall possibilities.

At Netscape, Tinderbox was a way of life (as it remains for Mozilla and many other development projects). The state of the tree was closely monitored, and acted as the focal point of development. If the tree was red, you could not checkin to the tree until it turned green. After you checked in your changes, you were added to a group called “the hook,” and as a member of the hook, you were required to watch the Tinderbox and ensure that the your changes built cleanly (that is, were green) for each platform affected. (Obviously, if the change was only made to, say, Mac-specific code, you were only obligated to see the Mac builds go green.) If the tree affected by your changes was green when you checked in, and then it went red in the first build that followed, then, as a member of the hook, you were required to help identify and fix the problem.

In addition to the hook, a *sheriff* was assigned by Netscape (or Mozilla) to monitor the overall state of the Tinderbox monitoring the tier-1 platforms, and to ensure that the builds all remained green. We eventually rotated the responsibility of sherifing inside of Netscape among the various development teams, one day the responsibility would fall to members of the mail-news team, on another, it was the responsibility of the IM team, and so on. Should a problem arise, the sheriff had the power to close the tree to all check-ins (this was communicated by a simple message at the top of the Tinderbox page), which was done in an attempt to aid those trying to isolate the cause of problems. The sheriff also had the authority to contact anyone on the hook, usually by e-mail, but by phone if necessary, should a diagnosis of the problem indicate that the person being contacted was to blame for the tree going red. As sheriff, I recall numerous times calling people by phone who had left the tree burning and had gone home for the night. Not everyone was happy about the policy, but it did cause people to be more careful about their work.

In a nutshell, Tinderbox plays an important role in cross-platform development because it forces developers to confront issues that affect the portability of code being committed to the repository. Although a responsible developer will try to determine the impact of changes on other platforms before landing code in the tree, this is not always done. Tinderbox acts as a backstop, ensuring that nothing slips through the cracks. And when problems are detected, either because of a red tree or a failed QA test the next day, Tinderbox can be used as an aid in determining which changes had been made that might be the cause of the problem.

Getting a green build on all the platforms is, of course, not the end of the story. A green build does not ensure that cross-platform feature parity is met for the platform, for example. A developer could implement a cross-platform feature and check in a full implementation for, say, Mac OS X and only stub implementations for Linux and Windows, and Tinderbox will not make this fact evident. What Tinderbox does do a good job of is ensuring that code shared among platforms builds everywhere, and that no one platform can hijack the stability of the tree (that is, leave the repository in a state such that it builds cleanly for only a subset of the supported platforms). As such, Tinderbox helps ensure that when platforms are worked on at the same time, the cross-platform train is moving ahead at the same pace for all platforms involved. This in turn helps ensure that the organization will be able to release a product to the market for all the supported platforms at about the same point in time, which is something that we

strived for at Netscape. Remember, however, that Tinderbox is just an aid—only by testing the resulting builds carefully can you confirm that a product or feature has achieved cross-platform parity.

Item 13: Use CVS or Subversion to Manage Source Code

An SCM system does exactly what the name implies; it helps you manage source code and related resources. To appreciate why SCM is important, consider for a moment what life without it might be like. Assume that you are a developer working on a new project that will consist of C++ sources files, a Makefile, and perhaps some resources such as icons or images. Obviously, these items must live somewhere, and so on day one of the project, you create a directory on your local desktop system where these files will live, and write a few hundred lines of code, storing the results in that directory. After a few weeks of hacking, you come up with version 0.1 of your creation, and after some light testing, you decide that your creation is ready to be pushed out onto the Web, with the hope of generating some user feedback from a small community of users.

After a few weeks, your e-mail inbox has accumulated several feature requests from users, and perhaps a dozen or so bug reports (in addition to a ton of spam because you gave out your e-mail address to the public). You get busy implementing some of these features and fixing the worst of the bugs, and after a few more weeks, you are ready to post version 0.2. This process repeats itself for a few months, and before you know it, you are shipping version 1.0 to an even wider audience.

The 1.0 release is a success, but isn't without its share of problems. First off, users are beginning to report a nasty bug in a feature that was first introduced in version 0.8, and was working flawlessly until version 1.0 was released to the public. You are able to duplicate the bug in a release version of the 1.0 binary, but can't duplicate it in a debugger. In an attempt to understand the problem, you pour over the code in search of clues; but after numerous hours of looking, you realize that you have no idea what might have caused this bug to surface. About the only way you can think of identifying the cause of the problem is to determine what specific changes you made to the codebase that might have led to the bug's manifestation. However, all you have to work with is the 1.0 source code, and you have no way of identifying the changes you made between 0.9 and 1.0.

The second problem before you is a request. It turns out that you removed a feature that was present in version 1.0, and removing the feature

has angered a lot of your users, who are clamoring for it to be reinstated in version 1.1. However, the code for this feature no longer exists, having been deleted from the source code long ago.

It is these two situations that, in my experience, make the use of a source code management system a necessity, cross-platform or not, no matter how large the project is or how many developers are involved. A good source code management system will allow you to re-create, in an instance, a snapshot of the source tree at some point in the past, either in terms of a specific date and time, or in terms of a specific release version. It will also help you to keep track of where and when changes to the source code have been made, so that you can go back and isolate specific changes related to a feature or a bug fix.

Had the developer used a source control management system, he or she could have retrieved versions of the source code starting at 0.9 and used this to isolate exactly what change(s) to the source caused the bug to first surface. And, to retrieve the source code for the feature that was removed in 1.0, the developer could have used the source code management system to retrieve the code associated with the feature, and undo its removal (or reengineer it back into the current version of the source code).

The benefits of a source code management system increase significantly as soon as multiple developers are assigned to a project. The main benefits from the point of view of a multideveloper project are accountability and consistency. To see how these benefits are realized, I need to describe in more detail how a source code management system works. Earlier, I described how a developer will typically manage a body of source code, in the absence of a source code management system, in a directory, which is usually created somewhere on the developer's system where a single copy of the source is stored and edited. The use of a source code management system changes things dramatically, however. When using a source code management system, the source code for a project is maintained in something called a repository, which you can think of as being a database that stores the master copy of a project's source code and related files. To work with the project source code, a developer retrieves a copy of the source code from the repository. Changes made to the local copy of the source code do not affect the repository. When the developer is done making changes to his or her copy of the source, he or she submits the changes to the source code management system, which will update the master copy maintained in the repository. It is important to realize that the repository records only the changes made to the file, instead of a complete copy of the latest version.

By storing only changes, you can easily retrieve earlier versions of files stored in the repository. The date of the change, the name or the ID of the developer who made the change, and any comments provided by the developer along with the change are all stored in the repository along with the change itself. The implications for developer accountability should be obvious; at any time, you can query the source code management system for a log of changes, when they were made, and by whom. This is a great help in locating the source of bugs and who may have caused them.

The ability to attribute a change in the repository to a developer, bug, or feature is directly affected by the granularity of check-ins made by developers on the project. Frequent, small changes to the repository will increase the ability of a developer to use the source control management system to identify and isolate change, and will also help ensure that other developers on the project gain access to the latest changes in a timely manner. A good rule of thumb is to limit the number of bugs fixed by a check-in to the repository to one (unless there are multiple, related bugs fixed by the same change).

So, now that you know the basic ideas being using an SCM, let's talk briefly about the implications to portability. First off, using an SCM is not a magic pill that makes your project portable. Portability requires attention to a lot more than just a source code management system to happen. (If that were not the case, this book would not need to be written.) But, using a source code management system that is available on each of the platforms that your organization is supporting (or plans to support) is, in my view, a critical part of any successful cross-platform project. It does no one any good if only Windows developers are able to pull source code, but Linux and Macintosh developers are left without a solution, after all. Not only should the SCM software be available everywhere, it at least should support a "lowest common denominator" user interface that behaves the same on all platforms, and to me, that means that the user interface needs to be command line based (both CVS and Subversion [SVN] support a command-line interface).

Because cross-platform availability and a common user interface are requirements, there are only two choices for an SCM system that I can see at the time of writing this book: CVS and SVN. At Netscape, and at countless other places (open source or not), CVS is the SCM of choice. It has stood the test of time, and is capable. It has been ported nearly everywhere, and its user interface is command line based. A very close cousin of CVS is SVN. After using SVN in a professional project, I have come to the conclusion

that for the programmers using it, SVN is quite similar to CVS in terms of how one approaches it and the commands that it offers, so either would be a good choice. (It is not without its quirks, however.) In this book, when I refer to an SCM, I am referring to CVS, but I could have easily said the same thing about SVN.

Besides providing a location from which Tinderbox can pull sources (see Item 12) and its support for Windows, Mac OS X, and Linux, perhaps the most important contribution of CVS to cross-platform development is its ability to create diff (or patch) files. The implications to cross-platform development of patch files are detailed in Item 14; in the following paragraphs, I describe what a patch file is and how CVS can be used to create a patch file.

A diff file, or a patch, is created by executing the `cvs diff` command. For example, assume I have added a method called `GetAlignment()` to a file named `nsLabel.h` in the Mozilla source tree. By typing `cvs diff`, I can easily identify the lines containing changes that I made:

```
$ cvs diff
cvs server: Diffing .
Index: nsLabel.h
=====
RCS file: /cvsroot/mozilla/widget/src/gtk/nsLabel.h,v
retrieving revision 1.21
diff -r1.21 nsLabel.h
61a62
>  NS_IMETHOD GetAlignment(nsLabelAlignment *aAlignment);
71d71
<  GtkJustification GetNativeAlignment();
```

The preceding output tells us that a line was added around line 61 of the file `nsLabel.h`, and one was removed around line 71 of the file. I can take this output, mail it to others on my team, and ask them to review it for errors or comments before checking in the changes. I can also look at this patch and make sure that it contains only those changes that I intended to land in the repository. I can't stress how important `cvs diff` is as a tool for identifying inadvertent check-ins before they are made.

With a lot of changes, the default output format that is shown here can be difficult to understand. A better output would show context lines, and make it more obvious which lines were added to the source, and which lines were deleted. The `-u` argument to `cvs diff` causes it to generate a "unified" diff, as follows:

```

$ cvs diff -u
cvs server: Diffing .
Index: nsLabel.h
=====
RCS file: /cvsroot/mozilla/widget/src/gtk/nsLabel.h,v
retrieving revision 1.21
diff -u -r1.21 nsLabel.h
--- nsLabel.h      28 Sep 2001 20:11:17 -0000      1.21
+++ nsLabel.h      1 Feb 2004 02:47:21 -0000
@@ -59,6 +59,7 @@
     NS_IMETHOD SetLabel(const nsString &aText);
     NS_IMETHOD GetLabel(nsString &aBuffer);
     NS_IMETHOD SetAlignment(nsLabelAlignment aAlignment);
+    NS_IMETHOD GetAlignment(nsLabelAlignment *aAlignment);

     NS_IMETHOD PreCreateWidget(nsWidgetInitData *aInitData);

@@ -68,7 +69,6 @@

protected:
    NS_METHOD CreateNative(GtkObject *parentWindow);
-    GtkJustification GetNativeAlignment();

    nsLabelAlignment mAlignment;

```

The differences in the output are the inclusion of context lines before and after the affected lines, and the use of + and - to indicate lines that have been added, or removed, respectively, from the source. This format is generally much easier on everyone who must read the patch, and it is the format that I recommend you use. You can change the number of lines of context generated by `cvs diff` by appending a count after the `-u` argument. For example, to generate only one line of context, issue the following command:

```

$ cvs diff -u1
cvs server: Diffing .
Index: nsLabel.h
=====
RCS file: /cvsroot/mozilla/widget/src/gtk/nsLabel.h,v
retrieving revision 1.21
diff -u -1 -r1.21 nsLabel.h
--- nsLabel.h      28 Sep 2001 20:11:17 -0000      1.21
+++ nsLabel.h      1 Feb 2004 02:50:45 -0000
@@ -61,2 +61,3 @@

```

```
NS_IMETHOD SetAlignment(nsLabelAlignment aAlignment);  
+ NS_IMETHOD GetAlignment(nsLabelAlignment *aAlignment);  
  
@@ -70,3 +71,2 @@  
NS_METHOD CreateNative(GtkObject *parentWindow);  
- GtkJustification GetNativeAlignment();
```

Generally, you'll want to generate somewhere between three or five lines of context for patches of moderate complexity. I use `-u3` almost religiously, and it is the default number of lines for `svn diff` (which does context diffs by default, too). However, don't be surprised if developers working with your patch files ask for more lines of context.

Setting Up and Using CVS

For those of you who are interested, I describe the steps it takes to set up a CVS server on a Red Hat-based system. The steps I provide here are almost certainly going to be the same when executed on non-Red Hat systems, and may differ in certain ways on other UNIX-based systems, including Mac OS X. The work involved in getting a CVS server up and running is not terribly difficult, and can be done in a relatively short amount of time. You will need root access to the system upon which you are installing the server, and it will help to have a second system with a CVS client so that you can test the result.

That said, if doing system administration makes you nervous, or site policy disallows it, or you do not have root access, check with a local guru or your system administrator for help.

To start the process of getting a CVS server running, you need to download the source for CVS from the Internet, build it, and install it. I retrieved a nonstable version of CVS by downloading the file `cvs-1.11.22.tar.gz` from <http://ftp.gnu.org/non-gnu/cvs/source/stable/1.11.22>. You are probably best, however, grabbing the latest stable version you can find.

After you have unpacked the file, `cd` into the top-level directory (in my case, `cvs-1.11.22`), and enter the following to build and install the source:

```
$ ./configure  
$ make  
$ su  
$ make install
```

Next, while still logged in as root, you need to do some work so that the CVS server daemon executes each time the system is rebooted. The first step is to check to see whether entries like the following are located in `/etc/services`:

```
cvspserver 2401/tcp      # CVS client/server operations
cvspserver 2401/udp      # CVS client/server operations
```

If these lines don't exist, add them to `/etc/services` as shown. Next, you need to create a file named `cvspserver` in `/etc/xinetd.d` that contains the following:

```
service cvspserver
{
    socket_type = stream
    protocol   = tcp
    wait       = no
    user       = root
    passenv    = PATH
    server     = /usr/bin/cvs
    server_args = -f --allow-root=/usr/cvsroot pserver
}
```

Make sure the permissions of this file are `-rw-r--r--`, and that its group and owner are root. This is probably the default, but it doesn't hurt to check.

If you are not yet running the desktop graphical user interface (GUI), fire it up, and from the Red Hat Start menu, select System Settings, Users and Groups to launch the Red Hat User Manager.

In the dialog that is displayed, click the Add Group button and add a group named `cvsgroup`. Next, click the Add User button, and add a user named `cvsuser`. You will be asked to provide a password; enter in something you can remember, and when you are done, exit the Red Hat User Manager.

Back in a terminal, and still as root, enter the following:

```
# cd /usr
# mkdir cvsroot
# chmod 775 cvsroot
# chown cvsuser cvsroot
# chgrp cvsgroup cvsroot
```

The preceding commands create the root directory for the CVS server. The path `/usr/cvsroot` corresponds to the value used in the `server_args` field of the service aggregate that was created earlier in `/etc/xinetd.d`. The following commands create a `locks` directory below `cvsroot`:

```
# cd cvsroot
# mkdir locks
# chown cvsuser locks
# chgrp cvsadmin locks
```

Now that the directory exists for the repository, it is time to create the repository. You can do this by executing the `cvs init` command, as follows:

```
# cvs -d /usr/cvsroot init
```

The `-d` argument specifies the location of the repository.

Now that the repository has been created, change to your home directory (for example, `/home/syd`), and execute the following command, which will check out the `CVSR00T` module from the repository that was just created:

```
# cvs -d /usr/cvsroot checkout CVSR00T
cvs checkout: Updating CVSR00T
U CVSR00T/checkoutlist
U CVSR00T/commitinfo
U CVSR00T/config
U CVSR00T/cvswrappers
U CVSR00T/editinfo
U CVSR00T/loginfo
U CVSR00T/modules
U CVSR00T/notify
U CVSR00T/rcsinfo
U CVSR00T/taginfo
U CVSR00T/verifymsg
```

Next, `cd` into the `CVSR00T` directory that was created by the preceding command, and open up the file named `config` using your favorite editor. Make the contents of this file consistent with the following:

```
# Set this to "no" if pserver shouldn't check system
# users/passwords.
SystemAuth=no
```



```
# Put CVS lock files in this directory rather than
# directly in the repository.
#LockDir=/var/lock/cvs

# Set 'TopLevelAdmin' to 'yes' to create a CVS
# directory at the top level of the new working
# directory when using the 'cvs checkout' command.
TopLevelAdmin=yes

# Set 'LogHistory' to 'all' or 'TOFEWGCMAR' to log all
# transactions to the history file, or a subset as
# needed (ie 'TMAR' logs all write operations)
#LogHistory=TOFEWGCMAR

# Set 'RereadLogAfterVerify' to 'always' (the default)
# to allow the verifymsg script to change the log
# message. Set it to 'stat' to force CVS to verify
# that the file has changed before reading it. This can
# take up to an extra second per directory being
# committed, so it is not recommended for large
# repositories. Set it to 'never' (the previous CVS
# behavior) to prevent verifymsg scripts from changing
# the log message.
#RereadLogAfterVerify=always
```

After you have made changes to the config file, check it into CVS as follows:

```
# cvs commit
cvs commit: Examining .
Checking in config;
/usr/cvsroot/CVSR00T/config,v <-- config
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database
```

In the same directory, run the following command to create a password for each user for whom you want to grant access to the repository. Every time you add a new developer to the project, you need to update the passwd file as I am about to describe, and check the changes into the repository:

```
# htpasswd passwd syd
New password:
Re-type new password:
Adding password for user syd
```

Now, open the file `passwd` (which was just created). At the end of the password, append `:cvsuser`. The result should look something like this:

```
syd:B9TyxNZ11EKb6:cvsuser
```

Next, you must add the password file to the repository, and commit the result:

```
# cvs -d /usr/cvsroot add passwd
cvs add: scheduling file 'passwd' for addition
cvs add: use 'cvs commit' to add this file permanently
# cvs -d /usr/cvsroot commit
RCS file: /usr/cvsroot//CVSROOT/passwd,v
done
Checking in passwd;
/usr/cvsroot//CVSROOT/passwd,v <-- passwd
initial revision: 1.1
done
cvs commit: Rebuilding administrative file database
```

This should result in two files in `/usr/cvsroot/CVSROOT`, one named `passwd,v` and the other named `passwd`. If there is not a file named `passwd` in `/usr/cvsroot/CVSROOT` (this could happen because of a bug in CVS), return to the checked-out version of `CVSROOT` (for example, the one in your home directory), edit the file named `checkoutlist`, and add a line to the end of the file that contains the text `passwd`. Then, doing a `cvs commit` on the `checkoutlist` file will cause the `passwd` file in `/usr/cvsroot/CVSROOT` to appear.

Now all that is left is to make the modules. Each directory you create under `/usr/cvsroot` is, logically, a project that is maintained in the repository. You can organize the hierarchy as you see fit. Here, I create a project named `client`:

```
# cd /usr/cvsroot
# mkdir client
# chown cvsuser client
# chgrp cvsadmin client
```

Now that we have created the repository, added a project, and set up some users, we can start the CVS server daemon by kicking `xinetd`:

```
# /etc/init.d/xinetd restart
Stopping xinetd:          [ OK ]
Starting xinetd:          [ OK ]
```

To ensure that the CVS server is running, run the following command:

```
# netstat -a | grep cvs
```

If you see output like the following, everything is in order, and you can use the repository:

```
tcp          0      0 *:cvspserver  *: *      LISTEN
```

To test out the new server and repository, find another machine, open up a shell (or a GUI CVS client if you prefer), and then check out the project named `client`. In the following example, I am using a command-line CVS client, and the server is located on my local network at the IP address 192.168.1.102:

```
$ cvs -d :pserver:syd@192.168.1.102:/usr/cvsroot login
(Logging in to syd@192.168.1.102)
CVS password:
$ cvs -d :pserver:syd@192.168.1.102:/usr/cvsroot co \
client
cvs server: Updating client
```

There now should be a directory named `client` in the current directory. If you `cd` into the `client` directory, you should see the following contents:

```
$ cd client
$ ls
CVS
```

At this point, you can add files and directories to the project with `cvs add`, and commit them to the repository using `cvs commit`.

Item 14: Use Patch

The patch program is considered by some to be the prime enabler behind the success of open source software. To quote Eric Raymond, “The patch program did more than any other single tool to enable collaborative

development over the Internet—a method that would revitalize UNIX after 1990” (*The Art of UNIX Programming*, Addison-Wesley, 2003). Of course, it is hard to imagine patch taking all the credit; after all, what would development be without `vi` (1)? But still, there is a ring of truth in what he says.

In the open source community, at any given moment, on any given project, there are dozens, if not hundreds of developers, all working on some derivation of what is in currently on the tip (or branch) of some source code repository. All of them are working relatively blind to the changes that their counterparts are making to the same body of source code.

An Example

Integrating (and evaluating) the changes made to a shared body of source code in such an environment can be difficult and error prone without a tool like patch. To see how, consider a team of three developers (A, B, and C) all working from the tip of the repository. Developer B is the team lead, and his job is to perform code reviews for Developer A and C, and integrate their changes into the source tree once an acceptable code review has been obtained. He also does development on the same body of source code, because he owns the overall architecture.

Let’s say that Developer A finishes his work and is in need of a code review. To obtain the code review, Developer B needs to communicate his changes to Developer B. I’ve seen this done a few different ways over the years:

- Developer A copies and pastes the changes made to the source file(s) into a text file, and sends the result to Developer B. In addition, Developer A adds comments to the text file to describe what the changes are, and where in the original source file the changes were made (or Developer A e-mails this information separately to Developer B). This is perhaps the worst method of all for conducting a code review, for two reasons:
 1. Developer A may make a mistake and not copy and paste all the changes that were made, or miss entire source files that have modification. The omission of a single line of change can greatly affect the ability of a code reviewer to accurately perform his task. Worse yet, if the code reviewer is responsible for integrating the changes into the repository and changes were missed, the process will surely lead to bugs.

2. Even if all changes are copied and pasted by Developer A, there is a chance that context will be lost or incorrectly communicated. One way to counter this problem would be for Developer A to include extra lines above and below the code that actually changed, but this is a better job for a tool like `cvs diff`, which can generate a patch file that contains the needed lines of context.
- Developer A sends to Developer B copies of all the source files that were changed. This is better than sending a series of hand-constructed diffs, because Developer B can now take the source files and create a patch that correctly captures the changes made by Developer A, along with the context of those changes. If Developer A sends source files that are not being modified by Developer B, Developer B can simply use the diff program (not `cvs diff` or `svn diff`) to generate a patch file relative to his current working tree. If Developer A, however, sends changes that do affect files modified by Developer B, Developer B can either diff against his working tree to see the changes in the context of work he is performing, or Developer B can pull a new tree somewhere and generate a patch file from it. The actual method used is usually best determined by the code reviewer. The downsides of this method are as follows:
 1. It is error prone. (Developer A might forget to include source files that contain change.)
 2. It places a burden on the code reviewer to generate a patch file. The last thing you want to do on a large project is make more work for the code reviewer. Usually, a code reviewer is generally always struggling to keep up with not only his own development task, but with all the code review requests that are pouring in. Anything you can do to make his job easier will generally be appreciative (and may result in the code reviewer giving your requests a higher priority).
 - Developer A generates a patch file using `cvs diff` or `svn diff`, and sends it to the code reviewer. This is the best method because
 1. The changes are relative to Developer A's source tree.
 2. `cvs diff` won't miss any changes that were made, assuming that `cvs diff` is run at a high-enough level of the source tree. (There is one exception: new source files that have not been added to the repository, along with forgetting to pass the `-N` argument to `cvs diff` when creating a patch file [this is not a problem with `svn diff`, which automatically includes new source files in its diff output.])

After the code reviewer (Developer B) receives the patch from Developer A, he or she has a few options:

- Simply look at the patch file, performing the code review based on its contents alone. Most of the time, this is what I do, especially if the patch file is a unified diff (as it should always be), and if the changes do not intersect any that I am making.
- Apply the patch file to his local tree, build the result, and then perhaps test it. This can be helpful if Developer B would like to step through an execution of the code in a debugger, or to see that the patch builds correctly and without warnings. If Developer A has made changes to some of the source files that were modified by Developer B, Developer B can either
 1. Pull a new tree and apply the patch to it so that his or her changes are not affected.
 2. Use `cvcs diff` to generate a patch file that contains his own changes, and then attempt to apply the changes from Developer A into his source tree. This allows Developer B not only to see the changes made by Developer A, but also to see them in the context of the changes that he is making. When the code review has been completed, Developer B can continue working on his changes, and check both his and Developer B's changes in at a later time, or Developer B can have Developer A check in the changes, and then do a `cvcs` or `svn` update to get in sync.

The patch program is the tool used by a code reviewer to apply changes specified in a patch file to a local copy of the repository. In essence, if both you and I have a copy of the same source tree, you can use `cvcs diff` to generate a patch file containing changes you have made to your copy of the sources, and then I can use the patch program, along with your patch file, to apply those changes to my copy of the sources. The patch program tries very hard to do its job accurately, even if the copy of the sources the patch is being applied to have been changed in some unrelated way. The type of diff contained in the patch file affects the accuracy attained by the patch program; patch is generally more successful if it is fed a set of context diffs rather than normal diffs. The `cvcs diff -u3` syntax (unified diff with three lines of context) is enough to generate a patch file that gives a good result. (SVN by default generates unified diffs with three lines of context.)

Patch Options

The patch program has a number of options. (You can refer to the patch man page for more details.) However, in practice, the only option that matters much is the `-p` argument, which is used to align the absolute paths used in the patch file with the local directory structure containing the sources that the patch is being applied to. When you run `cv diff` to create a patch file, it is best to do it from within the source tree, at the highest level in the directory hierarchy necessary to include all the files that have changes. The resulting patch file will, for each file that has changes, identify the file with a relative path, and patch uses this relative path to figure out what file in the target directory to apply changes to. For example:

```
Index: layout/layout.cpp
=====
RCS file: /usr/cvsroot/crossplatform/layout/layout.cpp,v
retrieving revision 1.33
diff -u -3 -r1.33 layout.cpp
--- layout/layout.cpp    27 May 2006 09:31:47 -0000    1.33
+++ layout/layout.cpp    7 Jun 2006 10:43:22 -0000
@@ -327,7 +327,7 @@
     return document;
 }

-int main(int argc, char *argv[])
+int LayoutMain(int argc, char *argv[])
{
    int run, parse;
    char *src = NULL;
```

The first line in the preceding patch (the line prefixed with `Index:`) specifies the pathname of the file to be patched. Assuming that the patch is contained in a file named `patch.txt`, then, if the preceding patch file were copied to the same relative location in the target tree, then issuing the following command is sufficient for patch to locate the files that are specified in the patch file:

```
$ patch -p0 < patch.txt
```

The `-p` argument will remove the smallest prefix containing the specified number of leading slashes from each filename in the patch file, using the result to locate the file in the local source tree. Because the patch file was

copied to the same relative location of the target tree that was used to generate the patch file in the source tree, we must use `-p0` because we do not want patch to remove any portion of the path names when searching for files. If `-p1` were used (with the same patch file, located in the same place in the target tree), the pathname `layout/layout.cpp` would be reduced to `layout.cpp`, and as a result, patch would not be able to locate the file, because the file would not be located in the current directory. Copying the patch file down into the `layout` directory would fix this, but this could only be done if, and only if, the patch file affected only sources that were located in the `layout` directory, because the `-p0` is applied by patch to all sources represented in the patch file.

Dealing with Rejects

So much for identifying which files to patch. The second difficulty you may run into is rejects. If patch is unable to perform the patch operation, it will announce this fact, and do one of two things. Either it will generate a reject file, which is a filename in the same directory as the file being patched, but with a `.rej` suffix (for example, `bar.cpp.rej`), or it will place text inside of the patched file to identify the lines that it was unable to resolve. (The `-dry-run` option can be used to preview the work performed by patch. As the name implies, it will cause patch to do a “dry run” of the patch operation, to let you know if it will succeed, without actually changing any of the target files.)

If either of these situations happens, there are a few ways to deal with it. The first thing I would do is remove the original source file, re-pull it from the repository using `cvcs update`, and try to reapply the patch, in case I was applying the patch to a file that was not up-to-date with the tip. If this didn't work, I would contact the person who generated the patch and ask that person to verify that his or her source tree was up-to-date at the time the patch file was generated. If it was not, I would ask that person to run `cvcs update` on the file or files and generate a new patch file.

If neither of these strategies works, what happens next depends on the type of output generated by patch. If patch created a `.rej` file, I would open it and the source file being patched in an editor, and manually copy and paste the contents of the `.rej` file into the source, consulting with the author of the patch file in case there are situations that are not clear. If, on the other hand, patch inlined the errors instead of generating a `.rej` file, open the source that was patched and search for lines containing `<<<`. These lines (and ones containing `>>>`) delimit the original and new source changes that were in conflict. By careful inspection of the patch output, and perhaps some

consultation with the author of the patch, you should be able to identify which portions of the resulting output should stay, and which portions of the result need to go, and perform the appropriate editing in the file to come up with the desired final result.

Thankfully, problems like this happen only rarely. The two most common causes of conflict occur when a developer is accepting a patch that affects code that he or she has also modified, or the patch files are created against a different baseline. There is little to do to avoid the first case, other than better communication among developers to ensure that they are not modifying the same code at the same time. The second case is usually avoided when developers are conscientious about ensuring that their source trees (and patches) are consistent with the contents of the CVS repository. When this is done, problems are relatively rare, and if they do occur, usually are slight and easy to deal with.

Patch and Cross-Platform Development

Now that you have an idea of why patch is so important to open source software, and how to use it, I need to describe how patch can make developing cross-platform software easier. At Netscape, each developer had a Mac, PC, and Linux system in his or her cubicle, or was at least encouraged to have one of each platform. (Not all did, in reality.) Each developer, like most of us, tended to specialize in one platform. (There were many Windows developers, a group of Mac developers, and a small handful of Linux developers.) As a result, it would only be natural that each developer did the majority of his or her work on the platform of his or her choice.

At Netscape, to overcome the tendency for the Windows developers to ignore the Linux and Macintosh platforms (I'm not picking on just Windows developers; Macintosh and Linux developers at Netscape were just as likely to avoid the other platforms, too), it was required that each developer ensure that all changes made to the repository correctly built and executed on each of the primary supported platforms, not just the developer's primary platform. To do this, some developers installed Network File System (NFS) clients on their Macintosh and Windows machines, and then pulled sources from the repository only on the Linux machine, to which both Mac and Windows machines had mounts. Effectively, Linux was a file server for the source, and the other platforms simply built off that source remotely. (The build system for Netscape/Mozilla allowed for this by isolating the output of builds; see Item 12.) This allowed, for example, the Windows developers to do all of their work on Windows, walk over to the

Mac and Linux boxes, and do the required builds on those platforms, using the same source tree.

But what if NFS (or, Samba these days) was not available? Or, more likely, developers did not have all three platforms to build on (or if they did, have the skills needed to make use of them)? In these cases, the patch program would come to the rescue. Developers could create a patch file, for example, on their Windows machine, and then either copy it to the other machines (where a clean source tree awaited for it to be applied to), or they could mail it to a “build buddy” who would build the source for those platforms that the developer was not equipped to build for. (Macintosh build buddies were highly sought after at Netscape because most developers at Netscape did not have the desire, or the necessary skills, to set up a Macintosh development system; it was much easier to ask one of the Macintosh helpers to be a build buddy.)

Netscape’s policy was a good one, and patch was an important part of its implementation. The policy was a good one because, by forcing all check-ins to build and run on all three platforms at the same time, it made sure that the feature set of each of the three platforms moved forward at the about the same pace (see Item 1). Mozilla, Netscape, and today, Firefox, pretty much work the same on Mac, Windows, and Linux, at the time of release. The combination of `cvs diff`, which accurately captured changes made to a local tree, and `patch`, which accurately merged these changes into a second copy of the tree, played a big role in enabling this sort of policy to be carried out, and allows projects such as Mozilla Firefox to continue to achieve cross-platform parity.