Subversion: The Definitive Guide

by Ben Collins−Sussman,Brian W. Fitzpatrick, and C. Michael Pilato

Subversion: The Definitive Guide

by Ben Collins−Sussman,Brian W. Fitzpatrick, and C. Michael Pilato

Published (TBA)

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

This will be the start of the preface.

What is Subversion

Subversion comes from...

How This Book is Organized

Conventions used in this book

The source code examples are just that˙ examples. While they will compile with the proper compiler incantations, they are intended to illustrate the problem at hand. So don't be surprised if an example is lacking error checking code and other bits that you would expect from production−quality code.

Comments and Questions

Insert ORA boilerplate here

Acknowledgements

Chapter 1. Introduction

Revision Control (and what svn can do for you)

TODO write me

Target audience

The intended audience of this book is anyone who has used a version control system before, although perhaps not Subversion or CVS. It assumes that the reader is computer–literate,and reasonably comfortable at a Unix command–line.

People familiar with CVS may want to skip some of the introductory sections that describe Subversion's concurrent versioning model. Also, there is a quick guide for CVS users attached as an appendix Appendix A

History

History of Revision Control

Subversion is a free/open–sourceversion control system. That is, Subversion manages files over time. The files are placed into a central repository. The repository is much like an ordinary file server, except that it remembers every change ever made to your files. This allows you to recover older versions of your files, or browse the history of how your files changed. Many people think of a version control system as a sort of ''time machine.''

Some version control systems are also software configuration management (SCM) systems. These systems are specifically tailored to manage trees of source code, and have many features that are specific to software development (such as natively understanding programming languages). Subversion, however, is not one of these systems; it is a general system that can be used to manage *any* sort of collection of files, including source code.

History of Subversion

Subversion aims to be the successor to the Concurrent Versions System (CVS). You can find more information about CVS at `http://www.cvshome.org/`.

At the time of writing, CVS is the standard Free version control system used by the open–sourcecommunity. It has a hard–earned,well–deservedreputation as stable and useful software, and has a design that makes it perfect for open–source development. However, it also has a number of problems that are difficult to fix.

Subversion'soriginal designers settled on a few simple goals. First, it was decided that Subversion should be a functional replacement for CVS. That is, it should do everything that CVS does, preserving the same development model while fixing the most obvious flaws. Secondly, with existing CVS users as the first target audience, Subversion should be written such that any CVS user should be able to start using it with little effort.

Collabnet `http://www.collab.net/` provided the initial funding in 2000 to begin development work, and the effort has now blossomed into a large, open–sourceproject backed by a community of free software developers.

Feature list (why svn is so nice)

What sort of things does Subversion do better than CVS? Here'sa short list to whet your appetite:

Advanced network layer
: The Subversion network server is Apache, and client and server speak WebDAV protocol to one another. ###TODO Xref to Designe

Faster network access
: A binary diffing algorithm is used to store and transmit deltas in both directions, re–

	gardless of whether a file is of text or binary type.
Meta−data	Each file or directory has an invisible hash table attached. You can invent and store any arbitrary key/value pairs you wish: owner, perms, icons, app−creator, mime−type, personal notes, etc. This is a general−purpose feature for users. Properties are versioned over time, just like file contents.
Hackability	Subversion has no historical baggage; it is primarily a collection of shared C libraries with well−defined APIs. This makes Subversion extremely maintainable and usable by other applications and languages.
Directory versioning	The Subversion repository doesn't use RCS files like CVS; instead, it implements a ˆvirtual˜ versioned filesystem that tracks changes to directory/file heirarchies over time. Files *and* directories are versioned. At last, there are real client−side **move** and **copy** commands.
Atomic commits	A commit either goes into the repository completely, or not all.

How to get svn binaries

TODO Write this.

How to get this book and send patches for it.

TODO Write this.

Chapter 2. Basic Concepts

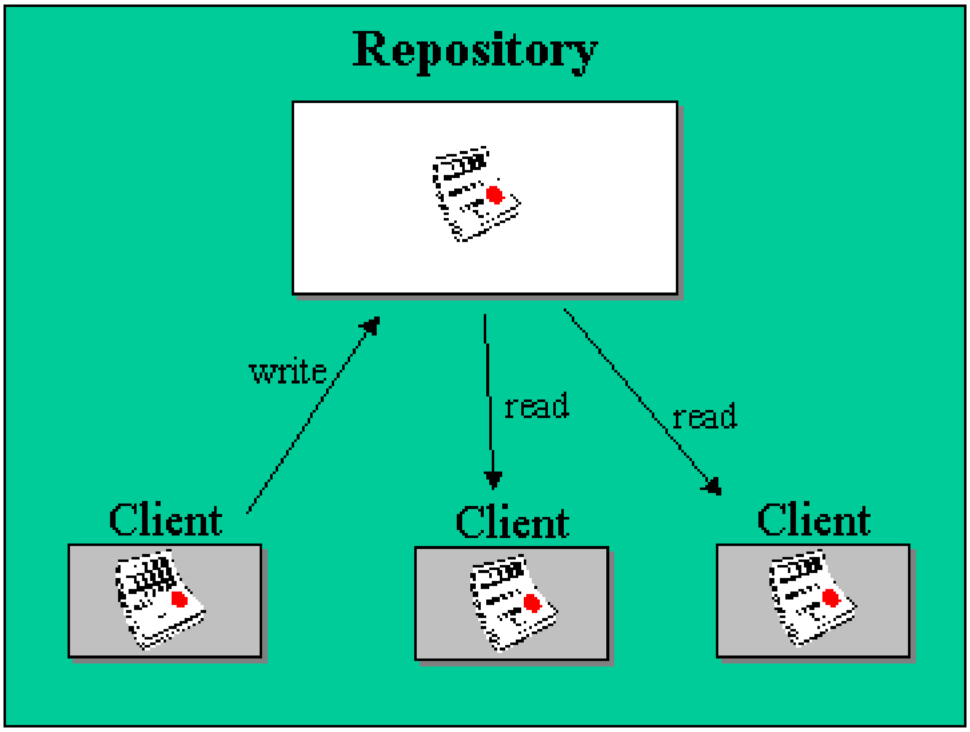
This chapter is a short, casual introduction to Subversion. If you're new to version control, this chapter is definitely for you. We begin with a discussion of general version control concepts, work our way into the specific ideas behind Subversion, and show some simple examples of Subversion in use.

Even though the examples in this chapter show people sharing collections of program source code, keep in mind that Subversion can manage any sort of file collection —it's not limited to helping computer programmers.

The Repository

Subversion is a centralized system for sharing information. At its core is a repository, which is a central store of data. The repository stores information in the form of a filesystem tree —a typical hierarchy of files and directories. Any number of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

Figure 2.1. A typical client/server system



So why is this interesting? So far, this sounds like the definition of a typical file server. And indeed, the repository *is* a kind of file server, but it's not your usual breed. What makes the Subversion repository special is that *it remembers every change* ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.

When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has the ability to view *previous* states of the filesystem. For example, a client can ask historical questions like, "what did this directory contain last Wednesday?", or "who was the last person to change this file, and what changes did they make?" These are the sorts of questions that are at the heart of any version control system: systems that are designed to record and track changes to data over time.

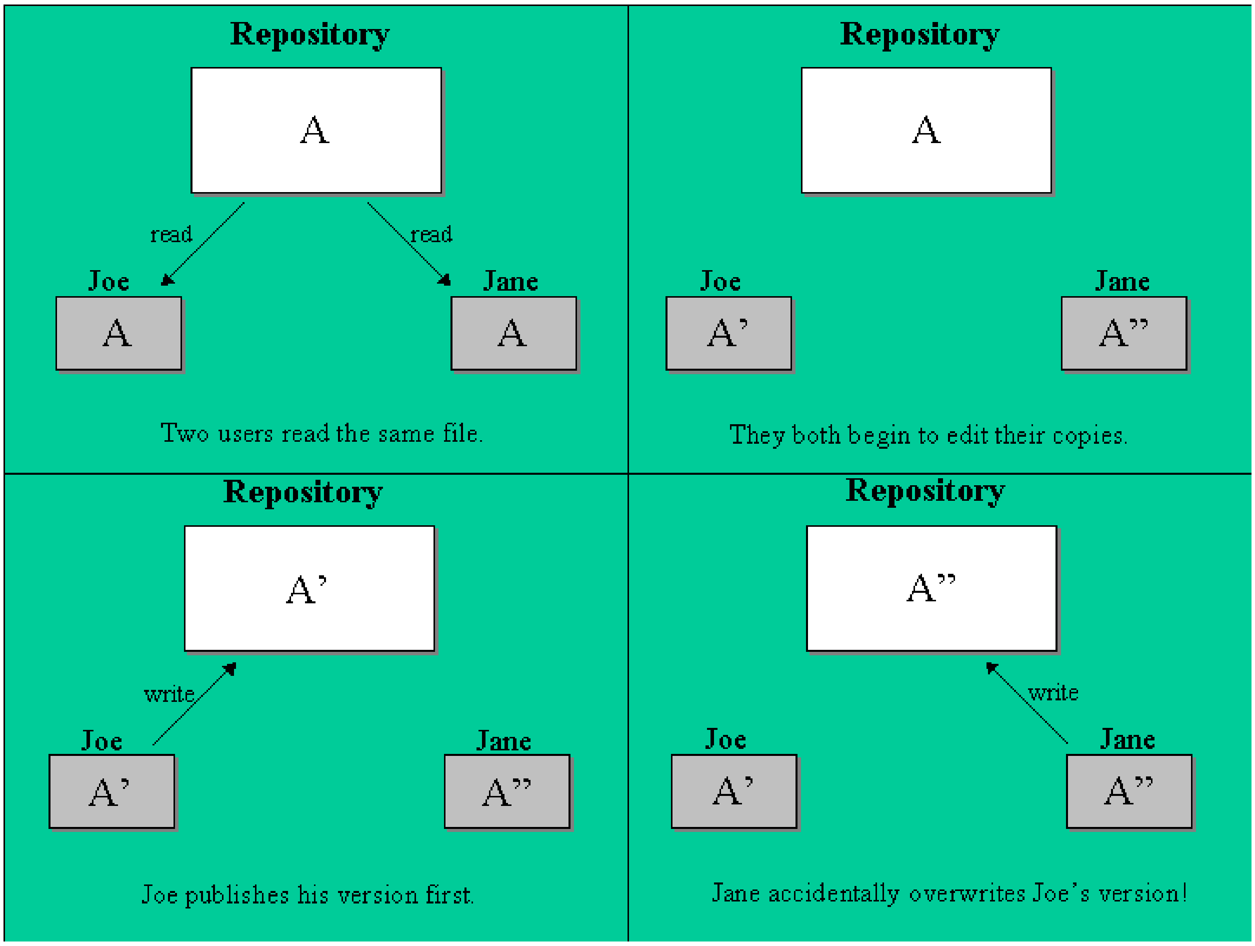
Versioning Models

The Problem of File–Sharing

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Consider this scenario: suppose we have two co–workers, Jane and Joe. They each decide to edit the same repository file at the same time. If Joe saves his changes to the repository first, then it's possible that (a few moments later) Jane could accidentally overwrite them with her own new version of the file. While Joe's version of the file won't be lost forever (because the system remembers every change), any changes Joe made *won't* be present in Jane's newer version of the file, because she never saw Joe's changes to begin with. Joe's work is still effectively lost ˙ or at least missing from the latest version of the file ˙ and probably by accident. This is definitely a situation we want to avoid!

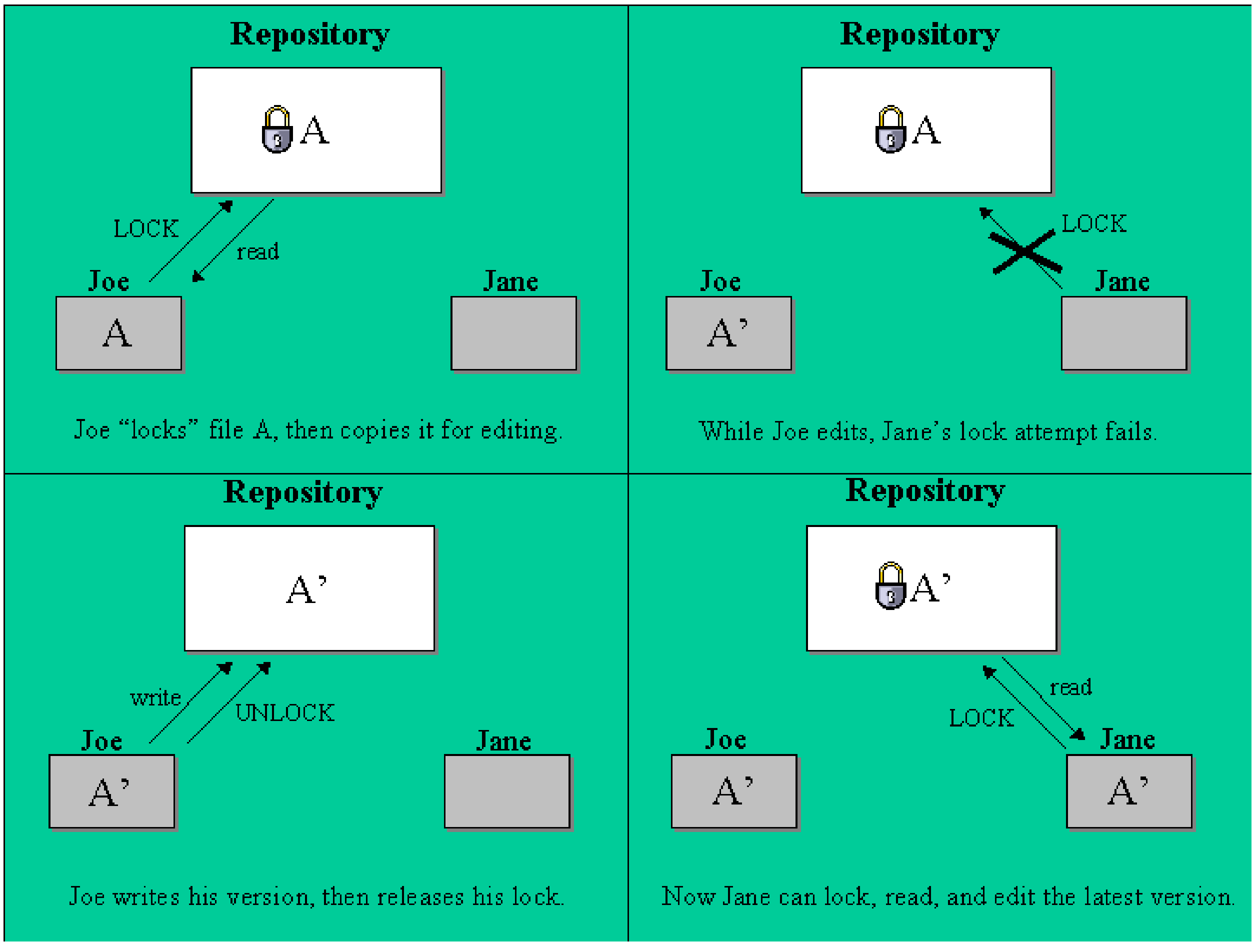
Figure 2.2. The problem to avoid



The Lock–Modify–Unlock Solution

Many version control systems use a lock–modify–unlock model to address this problem, which is a very simple solution. In such a system, the repository allows only one person to change a file at a time. First Joe must "lock" the file before he can begin making changes to it. Locking a file is a lot like borrowing a book from the library; if Joe has locked a file, then Jane cannot make any changes to it. If she tries to lock the file, the repository will deny the request. All she can do is read the file, and wait for Joe to finish his changes and release his lock. After Joe unlocks the file, his turn is over, and now Jane can take her turn by locking and editing.

Figure 2.3. The lock–modify–unlock solution



The problem with the lock–modify–unlock model is that it's a bit restrictive, and often becomes a roadblock for users:

- Sometimes Joe will lock a file and then forget about it. Meanwhile, because Jane is still waiting to edit the file, her hands are tied. And then Joe goes on vacation. Now Jane has to get an administrator to release Joe's lock. The situation ends up causing a lot of unnecessary delay and wasted time.
- What if Joe is editing the beginning of a text file, and Jane simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly merged together. There's no need for them to take turns in this situation.
- Pretend that Joe locks and edits file A, while Jane simultaneously locks and edits file B. But suppose that A and B depend on one another, and the changes made to each are semantically incompatible. Suddenly A and B don't work together anymore, and the locking system was powerless to prevent it˙ yet the locking system somehow provided a sense of false security, when it shouldn't have.

The Copy−Modify−Merge Solution

Subversion, CVS, and other version control systems use a copy−modify−merge model as an alternative to locking. In this model, each user's client reads the repository and creates a personal working copy of the file or project. Users then work in parallel, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly.

Here's an example. Say that Joe and Jane each create working copies of the same project, copied from the repository. They work concurrently, and make changes to the same file "A" within their copies. Jane saves her changes to the repository first. When Joe attempts to save his changes later, the repository informs him that his file A is out−of−date. In other words, that file A in the repository has somehow changed since he last copied it. So Joe asks his client to merge any new changes from the repository into his working copy of file A. Chances are that Jane's changes don't overlap with his own; so once he has both sets of changes integrated, he saves his working copy back to the repository.

Figure 2.4. The copy−modify−merge solution

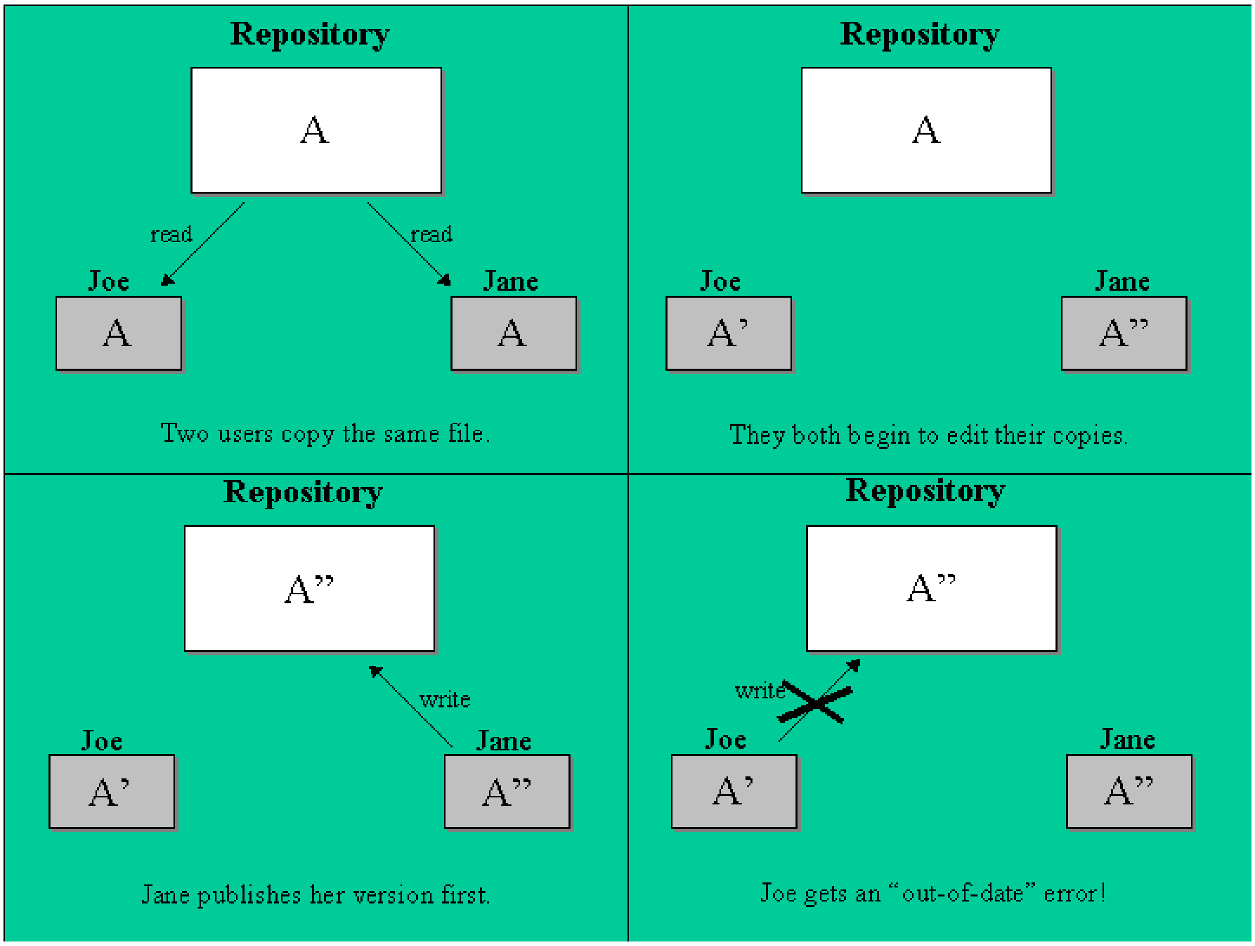
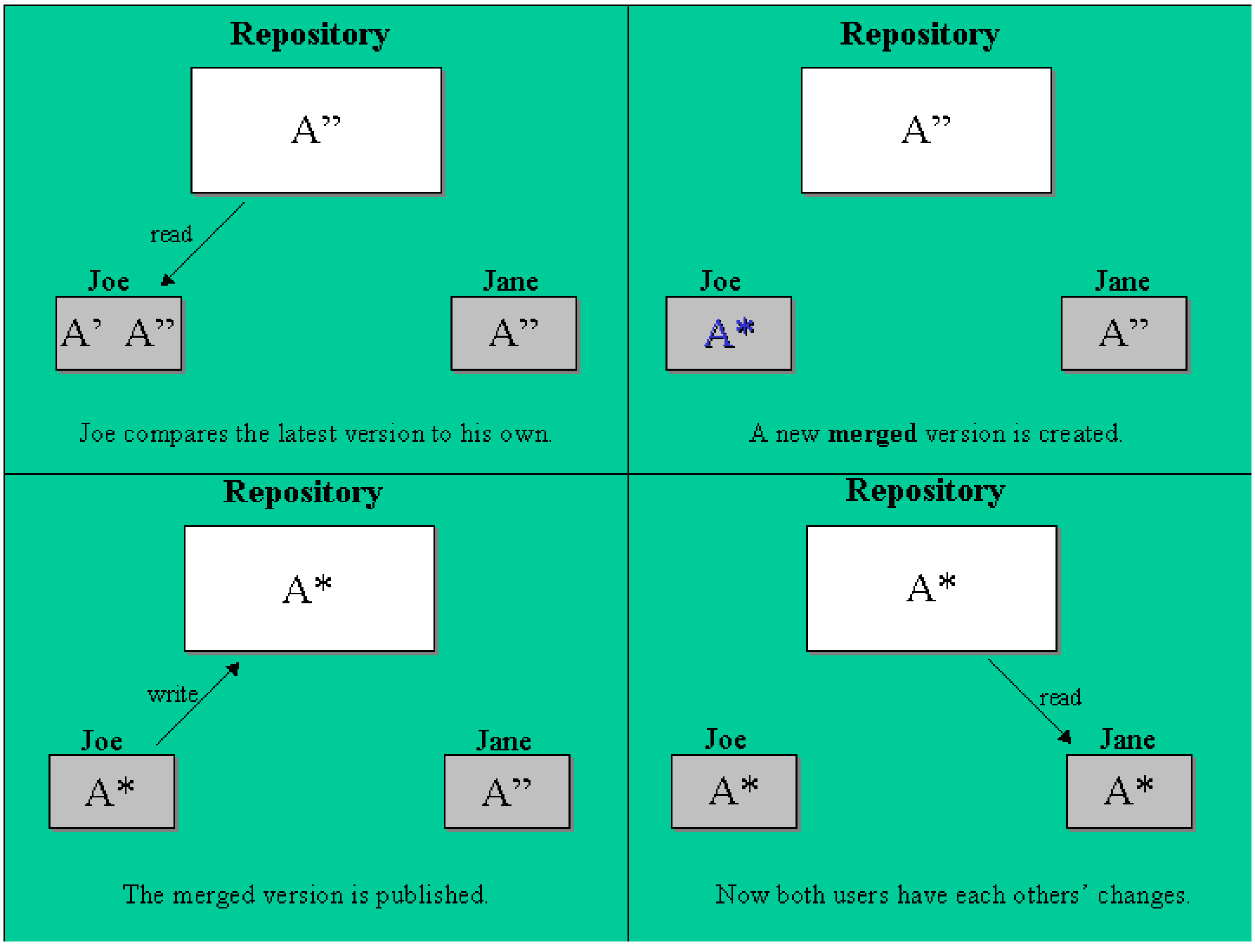


Figure 2.5. ...copy−modify−merge continued



But what if Jane's changes *do* overlap with Joe's changes? What then? This situation is called a conflict, and it's usually not much of a problem. When Joe asks his client to merge the latest repository changes into his working copy, his copy of file A is somehow flagged as being in a state of conflict: he'll be able to see both sets of conflicting changes, and manually choose between them. Note that software can't automatically resolve conflicts; only humans are capable of understanding and making intelligent choices. Once Joe has manually resolved the overlapping changes (perhaps by discussing the conflict with Jane!), he can safely save the hand–merged file back to the repository.

The copy–modify–merge model may sound a bit chaotic, but in practice, it runs extremely smoothly. Users can work in parallel, never waiting for one another. When they work on the same files, it turns out that most of their concurrent changes don't overlap at all; conflicts are infrequent. And the amount of time it takes to resolve conflicts is far less than the time lost by a locking system.

In the end, it all comes down to one critical factor: user communication. When users communicate poorly, both syntactic and semantic conflicts increase. No system can force users to communicate perfectly, and no system can detect semantic conflicts. So there's no point in being lulled into a false promise that a locking system will somehow prevent conflicts; in practice, locking seems to inhibit productivity more than anything else.

Subversion in action

Working copies

You've already read about working copies; now we'll demonstrate how the Subversion client creates and uses them.

A Subversion working copy is an ordinary directory tree on your local system, containing a collection of files. You can edit these files however you wish, and if they're source code files, you can compile your program from them in the usual way. Your working copy is your own private work area: Subversion will never incorporate other people's changes, nor make your own changes available to others, until you explicitly tell it to do so.

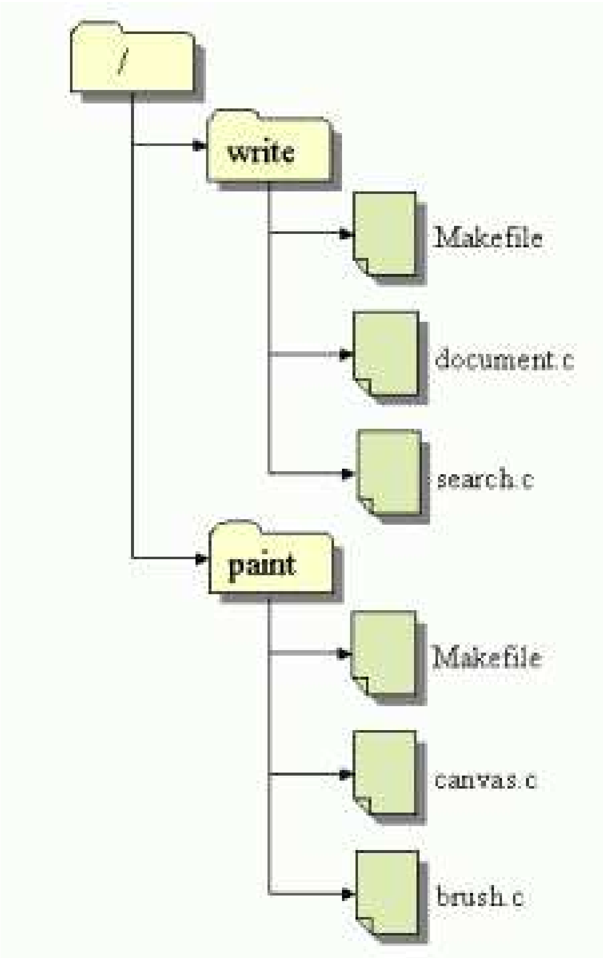
After you've made some changes to the files in your working copy and verified that they work properly, Subversion provides you with commands to "publish" your changes to the other people working with you on your project (by writing to the repository). If other people publish their own changes, Subversion provides you with commands to merge those changes into your working directory (by reading from the repository).

A working copy also contains some extra files, created and maintained by Subversion, to help it carry out these commands. In particular, each directory in your working copy contains a subdirectory named `.svn`, also known as the working copy administrative directory. The files in each administrative directory help Subversion recognize which files contain unpublished changes, and which files are out-of-date with respect to others' work.

A typical Subversion repository often holds the files (or source code) for several projects; usually, each project is a subdirectory in the repository's filesystem tree. In this arrangement, a user's working copy will usually correspond to a particular subtree of the repository.

For example, suppose you have a repository that contains two software projects.

Figure 2.6. The repository's filesystem



In other words, the repository's root directory has two subdirectories: `paint` and `write`.

To get a working copy, you must check out some subtree of the repository. (The term "check out" may sound like it has something to do with locking or reserving resources, but it doesn't; it simply creates a private copy of the project for you.) For example, If you check out `/write`, you will get a working copy like this:

```
$ svn checkout http://svn.example.com/repos/write
A  write
A  write/Makefile
A  write/document.c
A  write/search.c

$ ls -a write
Makefile  document.c  search.c  .svn/
```

Your working copy is a personal copy of the repository's `/write` directory, with one additional entry˙ `.svn`˙ which holds the extra information needed by Subversion, as mentioned earlier.

Suppose you make changes to `search.c`. Since the `.svn` directory remembers the file's modification date and original contents, Subversion can tell that you've changed the file. However, Subversion does not make your changes public until you explicitly tell it to. The act of publishing your changes is more commonly known as committing or (checking in) changes to the repository.

To publish your changes to others, you can use Subversion's **commit** command:

```
$ svn commit search.c
Sending search.c
Transmitting file data..
Committed revision 57.
```

Now your changes to `search.c` have been committed to the repository; if another user checks out a working copy of `/write`, they will see your changes in the latest version of the file.

Suppose you have a collaborator, Felix, who checked out a working copy of `/write` at the same time you did. When you commit your change to `search.c`, Felix's working copy is left unchanged; Subversion only modifies working directories at the user's request.

To bring his project up to date, Felix can ask Subversion to update his working copy, by using the Subversion **update** command. This will incorporate your changes into his working copy, as well as any others that have been committed since he checked it out.

```
$ pwd
/home/felix/write

$ ls -a
.svn/ Makefile document.c search.c

$ svn update
U search.c
```

The output from the **svn update** command indicates that Subversion updated the contents of `search.c`. Note that Felix didn't need to specify which files to update; Subversion uses the information in the `.svn` directory, and further information in the repository, to decide which files need to be brought up to date.

Revisions

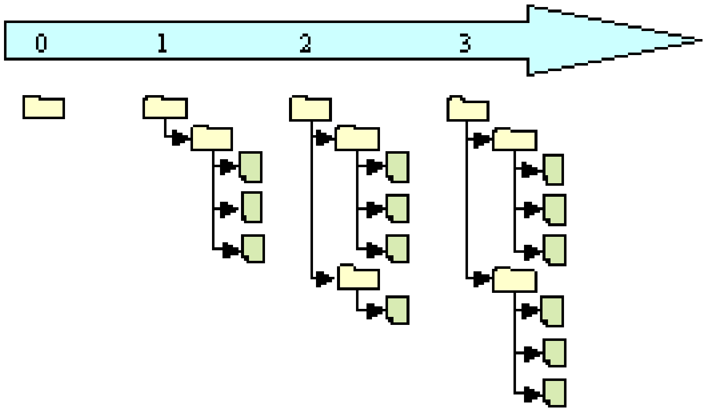
A **svn commit** operation can publish changes to any number of files and directories as a single atomic transaction. In your working copy, you can change files' contents, create, delete, rename and copy files and directories, and then commit the complete set of changes as a unit.

In the repository, each commit is treated as an atomic transaction: either all the commit's changes take place, or none of them take place. Subversion tries to retain this atomicity in the face of program crashes, system crashes, network problems, and other users' actions.

Each time the repository accepts a commit, this creates a new state of the filesystem tree, called a revision. Each revision is assigned a unique natural number, one greater than the number of the previous revision. The initial revision of a freshly created repository is numbered zero, and consists of nothing but an empty root directory.

A nice way to visualize the repository is as a series of trees. Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a snapshot of the way the repository looked after each commit.

Figure 2.7. The repository



Global revision numbers

Unlike those of many other version control systems, Subversion's revision numbers apply to *entire trees*, not individual files. Each revision number selects an entire tree, a particular state of the repository after some committed change. Another way to think about it is that revision N represents the state of the repository filesystem after the Nth commit. When a Subversion user talks about ‘‘revision 5 of `foo.c`’’,they really mean ‘‘`foo.c` as it appears in revision 5.’’Notice that in general, revisions N and M of a file do *not* necessarily differ! Because CVS uses per−filerevisions numbers, CVS users might want to look at Appendix A, "SVN for CVS Users", for more details.

It's important to note that working copies do not always correspond to any single revision in the repository; they may contain files from several different revisions. For example, suppose you check out a working copy from a repository whose most recent revision is 4:

```
write/Makefile:4
      document.c:4
      search.c:4
```

At the moment, this working directory corresponds exactly to revision 4 in the repository. However, suppose you make a change to `search.c`, and commit that change. Assuming no other commits have taken place, your commit will create revision 5 of the repository, and your working copy will now look like this:

```
write/Makefile:4
      document.c:4
```

```
       search.c:5
```

Suppose that, at this point, Felix commits a change to `document.c`, creating revision 6. If you use **svn update** to bring your working copy up to date, then it will look like this:

```
write/Makefile:6
       document.c:6
       search.c:6
```

Felix's changes to `document.c` will appear in your working copy, and your change will still be present in `search.c`. In this example, the text of `Makefile` is identical in revisions 4, 5, and 6, but Subversion will mark your working copy of `Makefile` with revision 6 to indicate that it is still current. So, after you do a clean update at the top of your working copy, it will generally correspond to exactly one revision in the repository.

How working copies track the repository

For each file in a working directory, Subversion records two essential pieces of information in the `.svn/` adminis– trative area:

- what revision your working file is based on (this is called the file's working revision), and
- a timestamp recording when the local copy was last updated by the repository.

Given this information, by talking to the repository, Subversion can tell which of the following four states a working file is in:

Unchanged, and current
: The file is unchanged in the working directory, and no changes to that file have been committed to the repository since its working revision. A **svn commit** of the file will do nothing, and a **svn update** of the file will do nothing.

Locally changed, and current
: The file has been changed in the working directory, and no changes to that file have been committed to the repository since its base revision. There are local changes that have not been committed to the repository, thus a **svn commit** of the file will succeed in publishing your changes, and a **svn update** of the file will do nothing.

Unchanged, and out–of–date
: The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated, to make it current with the public revision. A **svn commit** of the file will do nothing, and a **svn update** of the file fold the latest changes into your working copy.

Locally changed, and out–of–date
: The file has been changed both in the working directory, and in the repository. A **svn commit** of the file will fail with an "out–of–date" error. The file should be updated first; a **svn update** command will attempt to merge the public changes with the local changes. If Subversion can't complete the merge in a plausible way automatically, it leaves it to the user to resolve the conflict.

This may sound like a lot to keep track of, but as you will learn in a later chapter, the **svn status** command will show you the state of any item in your working copy.

Summary

We've covered a number of fundamental Subversion concepts in this chapter:

- We've introduced the notions of the central repository, the client working copy, and the array of repository revi–

sion trees.

- We've seen some simple examples of how two collaborators can use Subversion to publish and receive changes from one another, using the 'copy-modify-merge' model.

- We've talked a bit about the way Subversion tracks and manages information in a working copy.

At this point, you should have a good idea of how Subversion works in the most general sense. Armed with this knowledge, you should now be ready to jump into the next chapter, which is a detailed tour of Subversion's commands and features.

Chapter 3. Guided Tour

Now we will go into the details of using Subversion in your day to day work. By the time you reach the end of this chapter, you will be able to perform almost all the tasks you need to use Subversion in a normal day's work. You'll start with an initial checkout of your code, and walk through making changes and examining those changes. You'll also see how to bring changes made by others into your working copy, examine them, and work through any conflicts that might arise.

Note that this chapter is not meant to be an exhaustive list of all Subversion's commands˙ rather, it's a conversational introduction to the most common Subversion tasks you'll encounter. For a complete reference of all commands, see Chapter 8.

Help!

Before reading on, here is the most important command you'll ever need when using Subversion: **svn help**. The Subversion command-line client tries to be self-documenting˙ at any time, a quick **svn help <subcommand>** will describe the syntax, switches, and behavior of the **subcommand**.

Import

svn import is used to import a new project into a Subversion repository. While this is most likely the very first thing you will do when you set up your Subversion server, it's not something that happens very often. For a detailed description of import, see Chapter 5.

Initial Checkout

Most of the time, you will start using a Subversion repository by doing a checkout of your project. ˆChecking out˜ will provide you with a local copy of the `HEAD` (latest revision) of the Subversion repository that you specify on the command line.

```
$ svn co http://svn.collab.net/repos/svn/trunk
A  trunk/subversion.dsw
A  trunk/svn_check.dsp
A  trunk/COMMITTERS
A  trunk/configure.in
A  trunk/IDEAS

Checked out revision 2499.
```

Repository Layout

If you're wondering what `trunk` is all about in the above URL, it's part of the way that we recommend you lay out your Subversion repository. That is, when you create a new Subversion repository, you should make three top-level directories: `trunk`, `tags`, and `branches`. This may seem a little odd at the moment, but we'll talk a lot more about these in Chapter 5.

Although the above example checks out the trunk directory, you can just as easily checkout any deep subdirectory of a repository by specifying the subdirectory in the checkout URL:

```
$ svn co http://svn.collab.net/repos/svn/trunk/doc/book/tools
A  tools/readme-dblite.html
A  tools/fo-stylesheet.xsl
A  tools/svnbook.el
A  tools/dtd
```

```
A  tools/dtd/dblite.dtd

Checked out revision 3678.
```

Since Subversion uses a ˆcopy−modify−mergẽ model instead of ˆlock−modify−unlocķ, (See Chapter 2) you're now ready to start making changes to the files that you've checked out, known collectively as your working copy. You can even delete the entire working copy and forget about it˙ there's no need to notify the Subversion server unless you're ready to check in changes, a new file, or even a directory.

> **What's with the .svn directory?**
>
> Every directory in a working copy contains an administrative area, a subdirectory named `.svn`. Usually, directory listing commands won't show this subdirectory, but it is nevertheless an important directory. Whatever you do, don't delete or change anything in the administrative area! Subversion depends on it to manage your working copy.

While you can certainly check out a working copy with the url of the repository as the only argument, you can also specify a directory after your repository url. This places your working copy into the new directory that you name. For example:

```
$ svn co http://svn.collab.net/repos/svn/trunk subv
A  subv/subversion.dsw
A  subv/svn_check.dsp
A  subv/COMMITTERS
A  subv/configure.in
A  subv/IDEAS

Checked out revision 2499.
```

That will place your working copy in a directory named `subv` instead of a directory named `trunk` as was the case above.

Basic Workcycle

Subversion has numerous features, options, bells and whistles, but on a day−to−day basis, odds are that you will only use a few of them. In this section we'll run through the most common things that you might find yourself doing with Subversion in the course of a day's work.

The typical work cycle looks like this

- Update your working copy
- Make changes
- Examine your changes
- Merge others' changes
- Commit your changes

Update Your Working Copy

When working on a project with a team, you'll want to update your working copy: that is, receive any changes from other developers on the project. **svn update** brings your working copy in sync with the latest revision in the reposi− tory.

```
$ svn up
U  ./foo.c
U  ./bar.c
Updated to revision 2.
```

In this case, someone else checked in modifications to both `foo.c` and `bar.c` since the last time you updated, and Subversion has updated your working copy to include those changes.

Let's examine the output of **svn update** a bit more. When the server sends changes to your working copy, a letter code is displayed next to each item to let you know what actions Subversion performed to bring your working copy up to date:

`U foo`
: File `foo` was `U`pdated (received changes from the server).

`A foo`
: File or directory `foo` was `A`dded to your working copy.

`D foo`
: File or directory `foo` was `D`eleted from your working copy.

`R foo`
: File or directory `foo` was `R`eplaced in your working copy; that is, `foo` was deleted, and a new item with the same name was added. While they may have the same name, the repository considers them to be distinct objects with distinct histories.

`G foo`
: File `foo` received new changes from the repository, but your local copy of the file had changes that you made. The changes did not intersect, however, so Subversion has mer`G`ed the repository's changes into the file without a problem.

`C foo`
: File `foo` received `C`onflicting changes from the server. The changes from the server directly overlap your own changes to the file. No need to panic, though. This overlap needs to be resolved by a human (you); we discuss this situation later in this chapter.

Make Changes to Your Working Copy

Now you can to get to work and make changes in your working copy. It's usually most convenient to create a ˆtask˜ for yourself, such as writing a new feature, fixing a bug, etc. The Subversion commands that you will use here are **svn add**, **svn delete**, **svn copy**, and **svn move**. However, if you are merely editing a file (or files) that is already in Subversion, you may not need to use any of these commands until you commit.

Changes you can make to your working copy:

File changes
: This is the simplest sort of change. Unlike other version control systems, you don't need to tell Subversion that you intend to change a file; just make your changes. Subversion will be able to automatically detect which files have been changed.

Tree changes
: You can ask Subversion to ˆmark˜ files and directories for scheduled removal, addition, copying, or moving. While these changes may take place immediately in your working copy, no additions or removals will happen in the repository until you decide to commit.

To make file changes, just use your text editor, word processor, or graphics program˙ whatever tool you would normally use. A file needn't be in text–format; Subversion handles binary files just as easily as it handles text file (and just as efficiently too).

Here is an overview of the four Subversion subcommands that you'll use most often to make tree changes (we'll cover **svn import** and **svn mkdir** later).

svn add foo
: Schedule `foo` to be added to the repository. When you next commit, `foo` will become a permanent child of its parent directory. Note that if `foo` is a directory, only the directory itself will be scheduled for addition. If you want to add its contents as well, pass the `--recursive` (`-r`) switch.

svn delete foo
: Schedule `foo` to be deleted from the repository. If `foo` is a file, it is immediately deleted from your working copy˙ but it can be recovered with **svn revert** (discussed later). If `foo` is a directory, it is not deleted, but Subversion schedules it for deletion. When you commit your changes, `foo` will be removed from your working copy and the repository. 1

svn copy foo bar
: Create new item `bar` as a duplicate of `foo`. `bar` is automatically scheduled for addition. When `bar` is added to the repository on the next commit, its copy–historyis recorded (as having originally come from `foo`.)

svn move foo bar
: This command is exactly the same as running **svn cp foo bar; svn rm foo**. That is, `bar` is scheduled for addition as a copy of `foo`, and `foo` is scheduled for removal.

Changing the repository without committing

Earlier in this chapter, we said that you have to commit any changes that you make in order for the repository to reflect these changes. That's not entirely true˙ there *are* some use–cases that immediately commit tree changes to the repository. This only happens when a subcommand is operating directly on a URL, rather than on a working–copy path. (In particular, specific uses of **svn mkdir**, **svn cp**, **svn mv**, and **svn rm** can work with URLs).

URL operations behave in this manner because commands that operate on a working copy can use the working copy as a sort of ˆstaging area˜ to set up your changes before committing them to the repository. Commands that operate on URLs don't have this luxury, so when you operate directly on a URL, any of the above actions represent an immediate commit.

Examine Your Changes

svn status

Once you've finished making changes, you need to commit them to the repository, but before you do so, it's usually a good idea to take a look at exactly what you've changed. By examining your changes before you commit, you can not only make a more accurate log message, but you may discover that you've inadvertently changed a file, and this gives you a chance to revert those changes before committing. You can see exactly what changes you've made by using **svn status**, **svn diff**, and **svn revert** to find out what files have changed where and possibly remove these changes.

Subversion has been optimized to help you with this task, and is able to do many things without communicating with the repository. In particular, your working copy contains a secret cached ˆpristine˜ copy of each version controlled file within the `.svn` area. Because of this, Subversion can quickly show you how your working files have changed, or even allow you to undo your changes without contacting the repository.

You'll probably use the **svn status** command more than any other Subversion command.

CVS Users note

You're probably used to using **cvs update** to see what changes you've made to your working copy. **svn status** will give you all the information you need regarding what has changed in your working copy˙ without accessing the repository.

In Subversion, **update** does just that˙ it updates your working copy with any changes committed to the repository since the last time you've updated your working copy. You'll have to break the habit of using the **update** command to see what local modifications you've made.

If you run **svn status** at the top of your working copy with no arguments, it will detect all file and tree changes you've made. This example is designed to show all the different status codes that **svn status** can return. Note that the text in `[]` is not printed by **svn status**.

```
$ svn status
  L    ./abc.c               [svn has a lock in its .svn directory for abc.c]
M      ./bar.c               [the content in bar.c has local modifications]
 M     ./baz.c               [baz.c has property but no content modifications]
?      ./foo.o               [svn doesn't manage foo.o]
!      ./foo.c               [svn knows foo.c but a non-svn program deleted it]
~      ./qux                 [versioned as dir, but is file, or vice versa]
A  +   ./moved_dir           [added with history of where it came from]
M  +   ./moved_dir/README    [added with history and has local modifications]
D      ./stuff/fish.c        [this file is scheduled for deletion]
A      ./stuff/things/bloo.h [this file is scheduled for addition]
```

In this output format **svn status** prints four columns of characters followed by several whitespace characters followed by a file or directory name. The first column tells the status of a file or directory and/or its contents. The codes printed here are:

`file_or_dir`
: The file or directory has not been added or deleted, nor have `file_or_dir`'s contents been modified if it is a file. This line is printed only if you use **svn status −v**

`A file_or_dir`
: The file or directory `file_or_dir` has been scheduled for addition into the repository.

`M file`
: The contents of file `file` have been modified.

`D file_or_dir`
: The file or directory `file_or_dir` has been scheduled for deletion from the repository.

`? file_or_dir`
: The file or directory `file_or_dir` is not under version control. You can silence the question marks by either passing the `−−quiet` (`−q`) switch to **svn status**, or by setting the `svn:ignore` property on the parent directory.

  By default, **svn status** ignores files matching the regular expressions `*.o`, `*.lo`, `*.la`, `#*#`, `*.rej`, `*~`, and `.#*`. If you want additional files ignored, set the `svn:ignore` property on the parent directory. If you want to see the status of all the files in the repository regardless of **svn status** and **svn:ignore**'s regular expressions, then use the `−−no−ignore` command line option. See the section called ˆsvn:ignore˜ for more information.

`! file_or_dir`
: The file or directory `file_or_dir` is under version control but is missing from the working copy. This happens if the file or directory is removed using a non−Subversion command. A quick **svn up** or **svn revert file_or_dir** will restore the missing file from its cached pristine copy (directories must be refetched from the repository since the pristine copies are kept *inside* of the directory.

`~ file_or_dir`
: The file or directory `file_or_dir` is in the repository as one kind of object, but what's actually in your working copy is some other kind. For example, Subversion might have a file in the repository, but you removed the file and created a directory in

its place, without using the **svn delete** nor **svn add** commands.

The second column tells the status of a file or directory's properties. If an `M` appears in the second column, then the properties have been modified, otherwise a whitespace will be printed.

The third column will only show whitespace or an `L` which means that Subversion has locked the item in the `.svn` working area. You will see an `L` if you run **svn status** in a directory where an **svn commit** is in progress˙ perhaps when you are editing the log message. If Subversion is not running, then presumably Subversion was forcibly quit or died and the lock needs to be cleaned up by running **svn cleanup** (more about that later in this chapter). Locks typically appear if a Subversion command is interrupted before completion.

The fourth column will only show whitespace or a + which means that the file or directory is scheduled to be added or modified with additional attached history. This typically happens when you **svn move** or **svn copy** a file or directory. If you see `A    +`, this means the item is scheduled for addition−with−history. It could be a file, or the root of a copied directory. ˆ `+`˜ means the item is part of a subtree scheduled for addition−with−history, i.e. some parent got copied, and it's just coming along for the ride. `M    +` means the item is part of a subtree scheduled for addition−with−history, *and* it has local modifications. When you commit, first the parent will be added−with−history (copied), which means this file will automatically exist in the copy. Then the local modifications will be uploaded into the copy.

If you pass a specific path to **svn status**, it gives you information about that item alone:

```
$ svn status stuff/fish.c
D      stuff/fish.c
```

svn status also has a `--verbose` (`-v`) switch, which will show you the status of *every* item in your working copy, even if it has not been changed in your working copy:

```
$ svn status --verbose
M               44        23    sally     ./README
                44        30    sally     ./INSTALL
M               44        20    harry     ./bar.c
                44        18    ira       ./stuff
                44        35    harry     ./stuff/trout.c
D               44        19    ira       ./stuff/fish.c
                44        21    sally     ./stuff/things
A                0         ?     ?        ./stuff/things/bloo.h
                44        36    harry     ./stuff/things/gloo.c
```

This is the ˆlong form˜ output of **svn status**. The first column remains same, but the second column shows the working−revision of the item. The third and fourth columns show the revision in which the item last changed, and who changed it.

None of the above invocations to **svn status** contact the repository, they work only locally by comparing the metadata in the `.svn` directory with the working copy.

Finally, there is the `--show-updates` (`-u`) switch, which contacts the repository and adds information about things that are out−of−date:

```
$ svn status --show-updates --verbose
M      *        44        23    sally     ./README
M               44        20    harry     ./bar.c
       *        44        35    harry     ./stuff/trout.c
D               44        19    ira       ./stuff/fish.c
A                0         ?     ?        ./stuff/things/bloo.h
```

Notice the two asterisks: if you were to run **svn update** at this point, you would receive changes to `README` trout.c. This tells you some very useful information˙ you'll need to update and get the server–changes on README before you commit, or the repository will reject your commit for being out–of–date(More on this subject later).

svn diff

Another way to examine your changes is with the **svn diff** command. You can find out *exactly* how you've modified things by running **svn diff** with no arguments, which prints out file changes in unified diff format:

```
$ svn diff
Index: ./bar.c
===================================================================
--- ./bar.c
+++ ./bar.c Mon Jul 15 17:58:18 2002
@@ -1,7 +1,12 @@
+#include <sys/types.h>
+#include <sys/stat.h>
+#include <unistd.h>
+
+#include <stdio.h>

int main(void) {
-  printf("Sixty-four slices of American Cheese...\n");
+  printf("Sixty-five slices of American Cheese...\n");
return 0;
}

Index: ./README
===================================================================
--- ./README
+++ ./README Mon Jul 15 17:58:18 2002
@@ -193,3 +193,4 @@
+Note to self:  pick up laundry.

Index: ./stuff/fish.c
===================================================================
--- ./stuff/fish.c
+++ ./stuff/fish.c  Mon Jul 15 17:58:18 2002
-Welcome to the file known as 'fish'.
-Information on fish will be here soon.

Index: ./stuff/things/bloo.h
===================================================================
--- ./stuff/things/bloo.h
+++ ./stuff/things/bloo.h  Mon Jul 15 17:58:18 2002
+Here is a new file to describe
+things about bloo.
```

The **svn diff** command produces this output by comparing your working files against the cached ˆpristine˜ copies within the .svn area. Files scheduled for addition are displayed as all added–text, and files scheduled for deletion are displayed as all deleted text.

Output is displayed in unified diff format. That is, removed lines are prefaced with a - and added lines are prefaced with a +. **svn diff** also prints filename and offset information useful to the **patch** program, so you can generate ˆpatches˜ by redirecting the diff output to a file:

```
$ svn diff > patchfile
```

You could, for example, email the patchfile to another developer for review or testing prior to commit.

svn revert

Now suppose you see the above diff output, and realize that your changes to README are a mistake; perhaps you ac–cidentally typed that text into the wrong file in your editor.

This is a perfect opportunity to use **svn revert**.

```
$ svn revert README
Reverted ./README
```

Subversion reverts the file to its pre–modified state by overwriting it with the cached ˆpristine˜ copy from the `.svn` area. But also note that **svn revert** can undo *any* scheduled operations˙ for example, you might decide that you don't want to add a new file after all:

```
$ svn status foo
?      foo

$ svn add foo
A         foo

$ svn revert foo
Reverted foo

$ svn status foo
?      foo
```

Or perhaps you mistakenly removed a file from version control:

```
$ svn status README
       README

$ svn delete README
D         README

$ svn revert README
Reverted README

$ svn status README
       README
```

Look Ma! No Network!

All three of these commands (**svn status**, **svn diff**, and **svn revert**) can be used without any network access (except for the `--show-updates` (`-u`) switch to status). This makes it easy to manage your changes–in–progress when you find yourself somewhere without a network connection (e.g. traveling on an airplane, riding on a commuter train, etc.).

Subversion manages this by keeping private caches of pristine versions of each versioned file inside of the `.svn` ad–ministrative areas. This allows Subversion to report˙ and revert˙ local modifications to those files *without network access*. CVS keeps no such cache, and as a result has to use the network layer for practically everything. This cache (called the "text–base") also allows Subversion to, during a commit, send the user's local modifications to the server as a compressed delta against the pristine version. Since at commit time CVS has only the user's edited version of a file, it has to send the *entire file* to the server in order to relay the local modifications. At first glance, this might not seem too bad, but imagine the repercussions if try to commit a 1 line change to a 400MB file!

Resolving conflicts (Merging others' changes)

We've already seen how **svn status –u** can predict conflicts. Suppose you run **svn update** and some interesting things occur:

```
$ svn update
U  ./INSTALL
G  ./README
C  ./bar.c
```

The `U` and `G` codes are no cause for concern; those files cleanly absorbed changes from the repository. The files marked with `U` contained no local changes but were `U`pdated with changes from the repository. The `G` stands for mer`G`ed, which means that the file had local changes to begin with, but the changes coming from the repository didn't overlap in any way.

But the `C` stands for conflict. This means that the changes from the server overlapped with your own, and now you have to manually choose between them.

Whenever a conflict occurs, your Subversion client does the following:

- Subversion prints a `C` during the update, and remembers that the file is ˆconflicted.˜
- Subversion places Conflict markers into the file, to visibly demonstrate the overlapping areas.
- For every conflicted file, Subversion places three extra files in your working copy:
  - `filename.*.mine` This is your file as it existed in your working copy before you updated your working copy˙ that is, without conflict markers. This file has your latest changes in it and nothing else.
  - `filename.*.oldrevision` This is the file that was the BASE revision before you updated your working copy. That is, it the file that you checked out before you made your latest edits.
  - `filename.*.newrevision` This is the file that your Subversion client just received from the server when you updated your working copy. This file corresponds to the HEAD revision of the repository.

  Here`*` represents some random digits that Subversion chooses, `oldrevision` is the revision number of the file in your `.svn` directory, and `newrevision` is the revision number of the repository HEAD.

For example, Sally makes changes to the file `sandwich` in the repository. Harry has just changed the file in his working copy and checked it in. Sally updates her working copy before checking in and she gets a conflict:

```
$ svn update
C  sandwich
Updated to revision 2.
$ ls -1
sandwich
sandwich.59528.00001.r2
sandwich.59524.00001.r1
sandwich.59532.00001.mine
$
```

At this point, Subversion will *not* allow you to commit the file `sandwich`until the three temporary files are re– moved.

```
$ svn ci --message "Add a few more things"
subversion/libsvn_client/commit.c:654: (apr_err=155015, src_err=0)
svn: A conflict in working copy obstructs the current operation
svn: Commit failed (details follow):
subversion/libsvn_client/commit_util.c:180: (apr_err=155015, src_err=0)
svn: Aborting commit: '/home/sally/svn-work/sandwich' remains in conflict.
```

If you get a conflict, you need to either (1) hand−merge the conflicted text (by examining and editing the conflict markers within the file), (2) copy one of the tmpfiles on top of your working file, or (3) run **svn revert <filename>** to throw away all of your local changes.

```
$ cp sandwich.59532.00001.mine sandwich
```

Once you've resolved the conflict, you need to let Subversion know by removing the three temporary files. (The **svn resolve** command, by the way, is a shortcut that does nothing but automatically remove the three temporary files for you.) When those files are gone, Subversion no longer considers the file to be in a state of conflict anymore.

```
$ svn resolve sandwich
Resolved conflicted state of sandwich
```

Now you're ready to check in your changes

```
$ svn ci --message "Add a few more things"
Sending        sandwich
Transmitting file data .
Committed revision 3.
```

Note that**svn resolve**, unlike most of the other commands we've dealt with in this chapter, requires an argument. In any case, you want to be careful and only run **svn resolve** when you're certain that you've fixed the conflict in your file˙ once the temporary files are removed, Subversion will even let you check in a file with conflict markers.

Commit your changes

Finally! Your edits are finished, you've merged all updates from the server, and you're ready to commit your changes to the repository.

The **svn commit** command sends all (or, if you specify files or directories, some) of your changes to the repository. When you commit a change, you need to supply a log message, describing your change. Your log message will be attached to the new revision you create. If your log message is brief, you may wish to supply it on the command line using the `--message` (or `-m`):

```
$ svn commit --message "Corrected number of cheese slices."
Sending        sandwich
Transmitting file data .
Committed revision 3.
```

However, if you've been composing your log message as you work, you may want to tell Subversion to get the mes−sage from a file by passing the filename with the `--file` switch:

```
svn ci --file logmsg
Sending        sandwich
Transmitting file data .
Committed revision 4.
```

If you fail to specify either the `--message` or `--file` switch, then Subversion will automatically launch your fa−vorite editor (As defined in the environment variable `$EDITOR`) for composing a log message.

The repository doesn't know or care if your changes make any sense as a whole; it only checks to make sure that no−body else has changed any of the same files that you did when you weren't looking. If somebody *has* done that, the entire commit will fail with a message informing you that one or more of your files is out−of−date:

```
$ svn commit --message "Add another rule"
Sending        rules.txt
subversion/libsvn_client/commit.c:655: (apr_err=160028, src_err=0)
svn: Transaction is out of date
svn: Commit failed (details follow):
subversion/libsvn_repos/commit.c:112: (apr_err=160028, src_err=0)
svn: out of date: ‘rules.txt’ in txn ‘g’
$
```

At this point, you need to run **svn update**, deal with any merges or conflicts that result, and attempt your commit again.

That covers the basic work cycle for using Subversion. There are many other features in Subversion that you can use to manage your repository and working copy, but you can get by quite easily using only the commands that we’ve discussed so far in this chapter.

Examining History

As we mentioned earlier, the repository is like a time machine. It keeps a record of every revision ever committed, and allows you to explore this history by examining previous versions of files and directories as well as the metadata that accompanies them. With a single Subversion command, you can check out (or restore an existing working copy) the repository exactly as it was at any date or revision number in the past. However, sometimes you just want to *peer into* the past instead of *going into* the past.

There are several commands that can provide you with historical data from the repository. **svn log** shows you broad information: log messages attached to revisions, and which paths changed in each revision. **svn diff**, on the other hand, can show you the specific details of how a file changed over time. Finally, **svn cat** can be used to retrieve any file as it existed in a particular revision number and display it on your screen.

svn log

To find out information about the history of a file or directory, use the **svn log** command. **svn log** will provide you with a record of who made changes to a file or directory, at what revision it changed, the time and date of that revision, and, if it was provided, the log message that accompanied the commit.

```
$ svn log
------------------------------------------------------------------------
rev 3:  sally | Mon, 15 Jul 2002 18:03:46 -0500 | 1 line

Added include lines and corrected # of cheese slices.
------------------------------------------------------------------------
rev 2:  harry | Mon, 15 Jul 2002 17:47:57 -0500 | 1 line

Added main() methods.
------------------------------------------------------------------------
rev 1:  sally | Mon, 15 Jul 2002 17:40:08 -0500 | 2 lines

Initial import
------------------------------------------------------------------------
```

Note that the log messages are printed in *reverse chronological order* by default. If you wish to see a different range of revisions in a particular order, or just a single revision, pass the `--revision` (`-r`) switch:

```
$ svn log --revision 5:19
  # shows logs 5 through 19 in chronological order
$ svn log -r 19:5
  # shows logs 5 through 19 in reverse order
$ svn log -r 8
```

You can also examine the log history of a single file or directory. For example:

```
$ svn log foo.c

$ svn log http://foo.com/svn/trunk/code/foo.c
```

These will display log messages *only* for those revisions in which the working file (or URL) changed.

If you want even more information about a file or directory, **svn log** also takes a `--verbose` (`-v`) switch. Because Subversion allows you to move and copy files and directories, it is important to be able to track path changes in the filesystem, so in verbose mode, **svn log** will include a list of changed-pathsin a revision in its output:

```
$ svn log -r 8 -v
------------------------------------------------------------------------
rev 8:  sally | 2002-07-14 08:15:29 -0500 | 1 line
Changed paths:
U /trunk/code/foo.c
U /trunk/code/bar.h
A /trunk/code/doc/README

Frozzled the sub-space winch.

------------------------------------------------------------------------
```

svn diff

We've already seen **svn diff** above˙ it displays file differences in unified diff format; it was used to show the local modifications made to our working copy before committing to the repository.

In fact, it turns out that there are *three* distinct uses of **svn diff**:

- Examine local changes
- Compare your working copy to the repository
- Compare repository to repository

Examining local changes

As we saw above, invoking **svn diff** with no switches will compare your working files to the cached ˆpristine˜ copies in the `.svn` area:

```
$ svn diff
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth closed
+Chew with your mouth closed
+Listen when others are speaking
$
```

Comparing working copy to repository

If a single `--revision` (`-r`) number is passed, then your working copy is compared to the specified revision in the repository.

```
$ svn diff --revision 3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 3)
+++ rules.txt (working copy)
@@ -1,4 +1,5 @@
 Be kind to others
 Freedom = Responsibility
 Everything in moderation
-Chew with your mouth closed
+Chew with your mouth closed
+Listen when others are speaking
$
```

Comparing repository to repository

If two revision numbers, separated by a colon, are passed via `--revision` (`-r`), then the two revisions are directly compared.

```
$ svn diff --revision 2:3 rules.txt
Index: rules.txt
===================================================================
--- rules.txt (revision 2)
+++ rules.txt (revision 3)
@@ -1,4 +1,4 @@
 Be kind to others
-Freedom = Chocolate Ice Cream
+Freedom = Responsibility
 Everything in moderation
 Chew with your mouth closed
$
```

Not only can you use **svn diff** to compare files in your working copy to the repository, but if you supply a URL argument, you can examine the differences between items in the repository without even having a working copy. This is especially useful if you wish to inspect changes in a file when you don't have a working copy on your local machine:

```
$ svn diff --revision 4:5 http://svn.red-bean.com/repos/example/trunk/text/rules.txt

$
```

svn cat

If you want to examine an earlier version of a file and not necessarily the differences between two files, you can use **svn cat**:

```
$ svn cat --revision 2 rules.txt
Be kind to others
Freedom = Chocolate Ice Cream
Everything in moderation
Chew with your mouth closed
$
```

You can also redirect the output directly into a file:

```
$ svn cat --revision 2 rules.txt > rules.txt.v2
$
```

Now you're probably wondering why we don't just use **svn update −−revision** to update the file to the older revision. Well, there are several reasons that we might prefer to use **svn cat**.

First, you may want to see the differences between 2 revisions of a file using an external diff program (perhaps a graphical one, or perhaps your file is in such a format that the output of unified diff is nonsensical). In this case, you'll need to grab a copy of the old revision, redirect it to a file, and pass both that and the file in your working copy to your external diff program.

Second, you may have local changes to the file and don't want to lose them.

Finally, sometimes it's easier to look at an older revision in its entirety as opposed to just the differences between it and another revision.

Revision ˆNumbers˜

As you've probably noticed by now, many Subversion commands can take a `--revision` (`-r`) switch. Here we de− scribe some special ways to specify revisions.

The Subversion client understands a number of revision keywords. These keywords can be used instead of integer arguments to the `--revision` switch, and are resolved into specific revision numbers by Subversion:

HEAD
: The latest revision in the repository.

BASE
: The ˆpristine˜ revision of an item in a working copy.

COMMITTED
: The last revision in which an item changed.

PREV
: The revision just *before* the last revision in which an item changed. (Technically, COMMITTED −1).

Here are some examples of revision keywords in action:

```
$ svn diff --revision PREV:COMMITTED foo.c
# shows the last change committed to foo.c

$ svn log --revision HEAD
# shows log message for the latest repository commit

$ svn diff --revision HEAD
# compares your working file (with local mods) to the latest version
# in the repository.

$ svn diff --revision BASE:HEAD foo.c
# compares your ˆpristine˜ foo.c (no local mods) with the
# latest version in the repository

$ svn log --revision BASE:HEAD
# shows all commit logs since you last updated

$ svn update --revision PREV foo.c
# rewinds the last change on foo.c.
# (foo.c's working revision is decreased.)
```

These keywords may not seem terribly important, but they do allow you to perform many common (and helpful) op− erations without having to look up specific revision numbers or remember the exact revision of your working copy.

Other Commands

svn cleanup

When Subversion modifies your working copy (or any information within `.svn`), it tries to do so as safely as possi– ble. Before changing anything, it writes its intentions to a logfile, executes the commands in the logfile, then re– moves the logfile (This is similar in design to a journaled filesystem). If a Subversion operation is interrupted (If you hit Control–C,or if the machine crashes, for example), the logfiles remain on disk. By re–executingthe logfiles, Sub– version can complete the previously started operation, and your working copy can get itself back into a consistent state.

And this is exactly what **svn cleanup** does: it searches your working copy and runs any leftover logs, removing locks in the process. If Subversion ever tells you that some part of your working copy is ˆlocked˜, then this is the command that you should run. Also, **svn status** will display an `L` next to locked items:

```
$ svn status
  L    ./somedir
M      ./somedir/foo.c

$ svn cleanup
$ svn status
M      ./somedir/foo.c
```

svn info

In general, we try to discourage you from directly reading the `.svn/entries` file used to track items in a Subver– sion working copy. Instead, you can get tons of information about an item in your working copy by using the **svn info** command to display most of the tracked information:

```
$ svn info rules.txt
Path: rules.txt
Name: rules.txt
Url: http://svn.red-bean.com/repos/example/trunk/text/rules.txt
Revision: 8
Node Kind: file
Schedule: normal
Last Changed Author: harry
Last Changed Rev: 7
Last Changed Date: 2002-11-15 23:03:54 -0500 (Fri, 15 Nov 2002)
Text Last Updated: 2002-11-16 08:48:04 -0500 (Sat, 16 Nov 2002)
Properties Last Updated: 2002-11-16 08:48:03 -0500 (Sat, 16 Nov 2002)
Checksum: 8sfaU+5dqyOgkhuSdyxGrQ==
```

svn import

The import command is a quick way to move an unversioned tree of files into a repository.

There are two ways to use this command:

```
$ svnadmin create /usr/local/svn/newrepos
$ svn import file:///usr/local/svn/newrepos mytree
Adding  mytree/foo.c
Adding  mytree/bar.c
Adding  mytree/subdir
Adding  mytree/subdir/quux.h
Transmitting file data....
Committed revision 1.
```

The above example places the contents of directory `mytree` directly into the root of the repository:

```
/foo.c
/bar.c
/subdir
/subdir/quux.h
```

If you give **svn import** a third argument, it will use the argument as the name of a new subdirectory to create within the URL.

```
$ svnadmin create /usr/local/svn/newrepos
$ svn import file:///usr/local/svn/newrepos mytree fooproject
Adding  mytree/foo.c
Adding  mytree/bar.c
Adding  mytree/subdir
Adding  mytree/subdir/quux.h
Transmitting file data....
Committed revision 1.
```

The repository would now look like this:

```
/fooproject/foo.c
/fooproject/bar.c
/fooproject/subdir
/fooproject/subdir/quux.h
```

svn export

The export command is a quick way to create an unversioned tree of files from a repository directory˙ a tree that looks like a typical working copy, but doesn't contain the `.svn` directories:

```
$ svn export file:///usr/local/svn/newrepos/fooproject
A  fooproject/foo.c
A  fooproject/bar.c
A  fooproject/subdir
A  fooproject/subdir/quux.h
Checked out revision 3.
```

The resulting directory will not contain any `.svn` administrative areas, and all property metadata will be lost. (Hint: don't use this technique for backing up; it's probably better for rolling source distributions.)

svn list

The **svn list** command shows you what files are in a repository directory without actually downloading the files to your local machine:

```
$ svn list http://svn.collab.net/repos/svn
README
branches/
clients/
tags/
trunk/
```

If you want a more detailed listing, pass the `--verbose` (`-v`) flag to get output like this.

```
$ svn list --verbose http://svn.collab.net/repos/svn
_     2755   kfogel     1331 Jul 28 02:07 README
_     2773  sussman        0 Jul 29 15:07 branches/
_     2769 cmpilato        0 Jul 29 12:07 clients/
_     2698   rooneg        0 Jul 24 18:07 tags/
_     2785    brane        0 Jul 29 19:07 trunk/
```

The columns tell you if a file has any properties (ˆP˜ if it does, ˆ_˜ if it doesn't), the revision it was last updated at, the user who last updated it, the size if it is a file, the date it was last updated, and the item's name.

svn mkdir

This is another convenience command, and it has two uses.

First, it can be used to simultaneously create a new working copy directory and schedule it for addition:

```
$ svn mkdir new-dir
A     new-dir
```

Or, it can be used to instantly create a directory in a repository (no working copy needed):

```
$ svn mkdir file:///usr/local/svn/newrepos/branches --message "made new dir"
Committed revision 1123.
```

Again, this is a form of immediate commit, so some sort of log message is required.

Now we've covered all of the Subversion client commands with the exception of those dealing with branching and merging (see Chapter 4) and properties (see the section called ˆProperties˜). However, you may want to take a moment to skim through Chapter 8 to get an idea of the many different ways that you can use the commands that we covered above.

Chapter 4. Branching and Merging

Branching, tagging, and merging are concepts common to almost all version control systems. If you're not familiar with these ideas, we provide a good introduction in this chapter. If you are familiar, then hopefully you'll find it interesting to see how Subversion implements these ideas.

Either way, branching is a fundamental part of version control. If you're going to allow Subversion to manage your data, then this is a feature you'll eventually come to depend on.

What's a branch?

A good way to explain the idea of a "branch" is with an example.

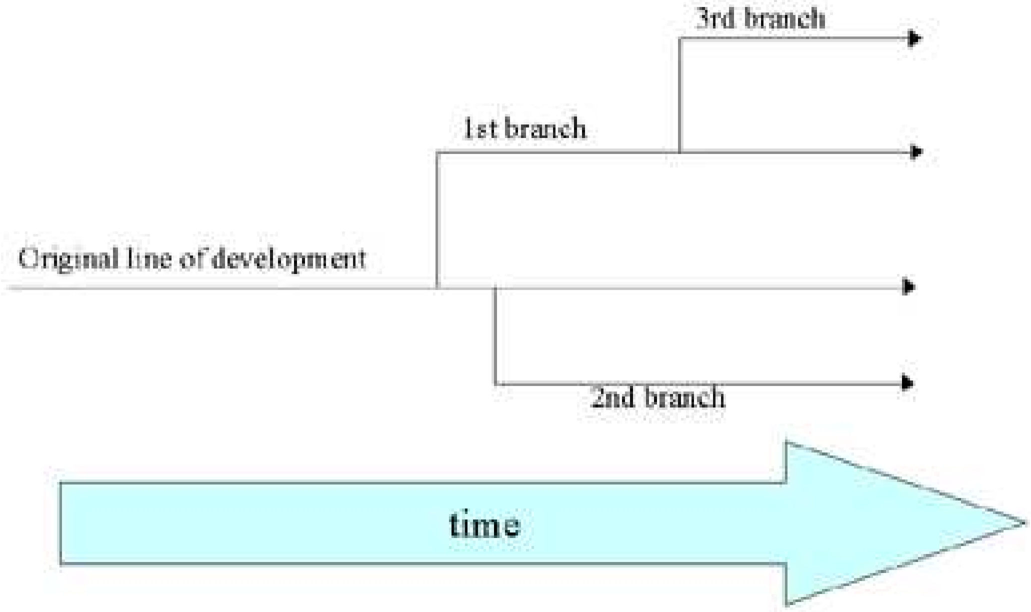
Suppose it's your job to maintain a document for a division in your company, a handbook of some sort. One day a different division asks you for the same handbook, but with a few parts 'tweaked' for them, since they do things slightly differently.

What do you do in this situation? You do the obvious thing: you make a second copy of your document, and begin maintaining the two copies separately. As each department asks you to make small changes, you incorporate them into one copy or the other.

Of course, often you want to make the same change to both copies. For example, if you discover a typo in the first copy, it's very likely that the same typo exists in the second copy. The two documents are almost the same, after all; they only differ in small, specific ways.

This is the basic concept of a branch ˙ namely, a line of development that exists independently of another line, yet still shares a common history if you look far enough back in time. A branch always begins life as a copy of something, and moves on from there, generating its own history.

Figure 4.1. Branches of development



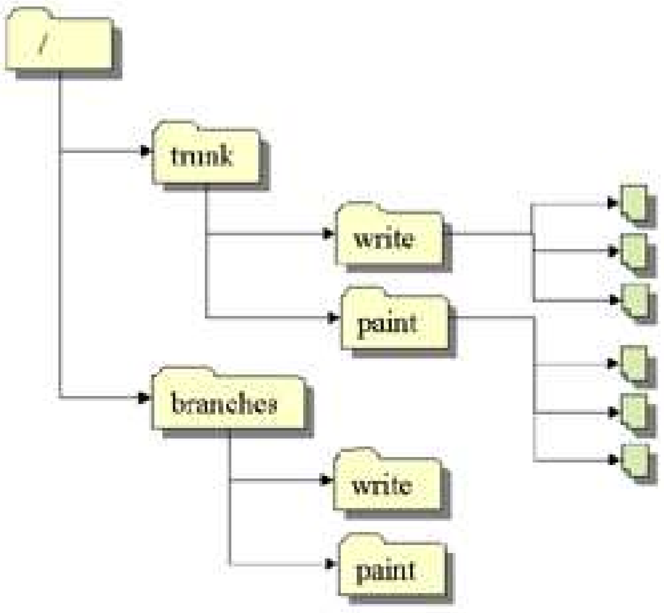
Subversion has commands to help you maintain parallel branches of your files and directories. It allows you to create branches by copying your data, and remembers that the copies are related to one another. It also helps you duplicate changes from one branch to another. Finally, it can make portions of your working copy reflect different branches, so that you can "mix and match" different lines of development in your daily work.

Using branches

At this point, you should understand how each commit creates an entire new filesystem tree (called a "revision") in the repository. If not, go back and read about revisions in the section called ˆRevisions˜.

For this chapter, we'll go back to the same example from Chapter 2. Remember that you and your collaborator, Felix, are sharing a repository that contains two projects, `paint` and `write`. Notice, however, that this time somebody has created two new top–level directories in the repository, called `trunk` and `branches`. The projects themselves are subdirectories of `trunk`, and the reason for this will become clearer later on.

Figure 4.2. Starting repository layout



As before, assume that you and Felix both have working copies of the `/trunk/write` project.

Let's say that you've been given the task of performing a radical reorganization of the project. Not only is it a wide change (it will affect all the files in the project), but it's a very large change (it will take a long time to write.) The problem here is that you don't want to interfere with Felix, who is in the process of fixing small bugs here and there. He's depending on the fact that the latest version of the project is always usable. If you start committing your changes bit–by–bit, you'll surely break things for Felix.

One strategy is to crawl into a hole: you and Felix can stop sharing information for a week or two. That is, start gutting and reorganizing all the files in your working copy, but don't commit or update until you're completely finished with the task. There are a number of problems with this, though. First, it's not very safe. Most people like to save their work to the repository frequently, should something bad accidentally happen to their working copy. Second, it's not very flexible. If you do your work on different computers (perhaps you have a working copy of `/trunk/write` on two different machines), you'll need to manually copy your changes back and forth, or just do all the work on a single computer. Finally, when you're finished, you might find it very difficult to commit your changes. Felix (or others) may have made many other changes in the repository that are difficult to merge into your working copy –– especially all at once.

The better solution, of course, is to create your own branch, or line of development, in the repository. This allows you to save your half–broken work frequently without interfering with others, yet you can still selectively share information with your collaborators. You'll see exactly how this works later on.

Creating a branch

So how do we create a branch? Very simple –– you make a copy of the project in the repository using the **svn copy** command. Subversion is not only able to copy single files, but whole directories as well. In this case, you want to make a copy of the `/trunk/write` directory. Where should the new copy live? It doesn't really matter; Subversion doesn't care. It's a matter of project policy. Let's say that your team has a policy of creating branches in the `/branches/write` area of the repository, and you want to name your branch "my–write–branch". So you want to create a new directory, `/branches/write/my-write-branch`, which starts as a copy of `/trunk/write`.

There are two different ways to make a copy. We'll demonstrate the messy way first, just to make the concept clear. To begin, check out a working copy of the root (`/`) of the repository:

```
$ svn checkout http://svn.example.com/repos bigwc
A  bigwc/branches/
A  bigwc/branches/write
A  bigwc/branches/paint
A  bigwc/trunk/
A  bigwc/trunk/write
A  bigwc/trunk/write/Makefile
A  bigwc/trunk/write/document.c
A  bigwc/trunk/write/search.c
A  bigwc/trunk/paint
A  bigwc/trunk/paint/Makefile
A  bigwc/trunk/paint/canvas.c
A  bigwc/trunk/paint/brush.c
Checked out revision 340.
```

And now it's simply a matter of giving two working–copy paths to the **svn copy** command:

```
$ cd bigwc
$ svn copy trunk/write branches/write/my-write-branch
$ svn status
A  +   branches/write/my-write-branch
```

In this case, the **svn copy** command recursively copied the `trunk/write` working directory to a new working directory, `branches/write/my-write-branch`. As you can see from the **svn status** command, the new directory is now scheduled for addition to the repository. But also notice the '+' next to the letter A. This indicates that the scheduled addition is a *copy* of something, not something new. When you commit your changes, Subversion will create `/branches/write/my-write-branch` in the repository by copying `/trunk/write`, rather than resending all of the working copy data over the network:

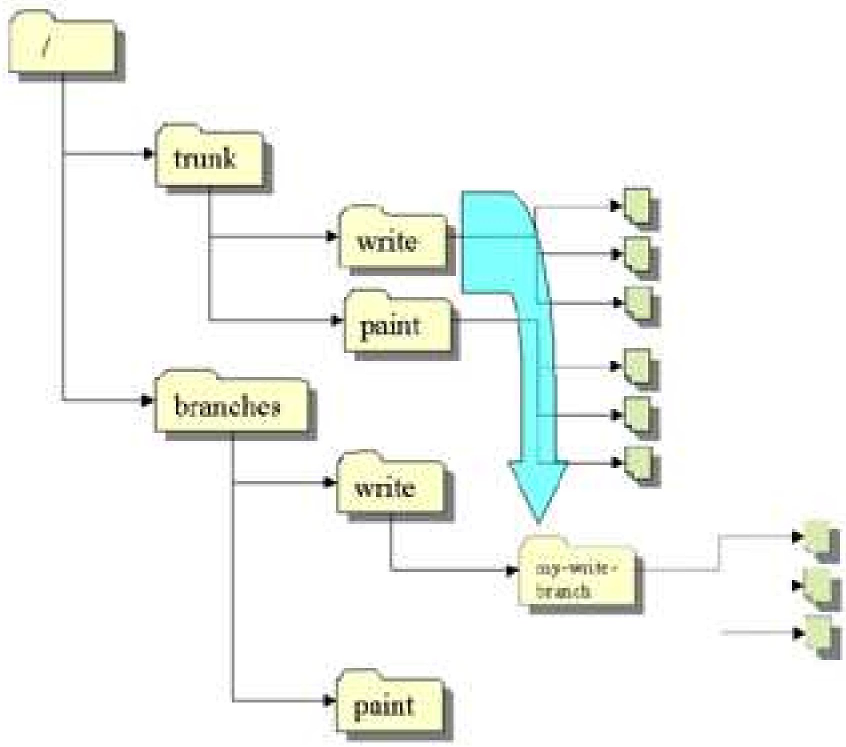
```
$ svn commit -m "Creating a private branch of /trunk/write."
Adding      branches/write/my-write-branch
Committed revision 341.
```

And now the easier method of creating a branch, which we should have told you about in first place: the **svn copy** is able to operate on two URLs.

```
$ svn copy http://svn.example.com/repos/trunk/write \
           http://svn.example.com/repos/branches/write/my-write-branch \
      -m "Creating a private branch of /trunk/write"

Committed revision 341.
```

There's really no difference between these two methods. Both procedures create a new directory in revision 341, and the new directory is a copy of `/trunk/write`. Notice that the second method, however, performs an *immediate* commit. It's an easier procedure, because it doesn't require you to check out a large mirror of the repository. In fact, this technique doesn't even require you to have a working copy at all!

Figure 4.3. Repository with new copy



> **Cheap copies**
>
> At this point, you might be thinking to yourself: "Holy cow, the repository is copying entire directories −−doesn't that mean that the repository will get really huge if you start making copies of big projects?"
>
> This is a valid concern! However, no need to worry. Subversion's repository has a clever internal design. When the repository copies something, it doesn't actually duplicate any data. Instead, it creates a new directory entry that points to an *existing* tree. If you're a Unix user, this is the same concept as a hard−link.From there, the copy is said to be "lazy". That is, if you commit a change to one file within the copied directory, then only that file changes −−the rest of the files continue to exist as links to the original files in the original directory.
>
> This is why you'll often hear Subversion users talk about "cheap copies". It doesn't matter how large the directory is −−it takes a very tiny, constant amount of time to make a copy of it. In fact, this feature is the basis of how commits work in Subversion: each revision is a "cheap copy" of the previous revision, with a few items lazily changed within. To read more about this, look at Subversion's Design document.
>
> The point here is that copies are cheap, both in time and space. Make branches as early and often as you want.

Working with your branch

Now that you've created a new branch of the project, you can check out a new working copy to start using it:

```
$ svn checkout http://svn.example.com/repos/branches/write/my-write-branch
A  my-write-branch/Makefile
A  my-write-branch/document.c
A  my-write-branch/search.c
Checked out revision 341.
```

There's nothing special about this working copy; it simply mirrors a different location of the repository. When you commit changes, however, Felix won't ever see them when he updates. His working copy is of `/trunk/write`.

Let's pretend that a week goes by, and the following commits happen:

- You make a change to `/branches/write/my-write-branch/search.c`, which creates revision 342.
- You make a change to `/branches/write/my-write-branch/document.c`, which creates revision 343.
- Felix makes a change to `/trunk/write/document.c`, which creates revision 344.

Things get interesting when you look at the history of changes made to your copy of `document.c`:

```
$ pwd
/home/user/my-write-branch

$ svn log document.c
------------------------------------------------------------------------
rev 343:  user | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines

* document.c:  frozzled the wazjub.

------------------------------------------------------------------------
rev 303:  felix | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines

* document.c:  changed a docstring.

------------------------------------------------------------------------
rev 98:  felix | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines

* document.c:  adding this file to the project.

------------------------------------------------------------------------
```

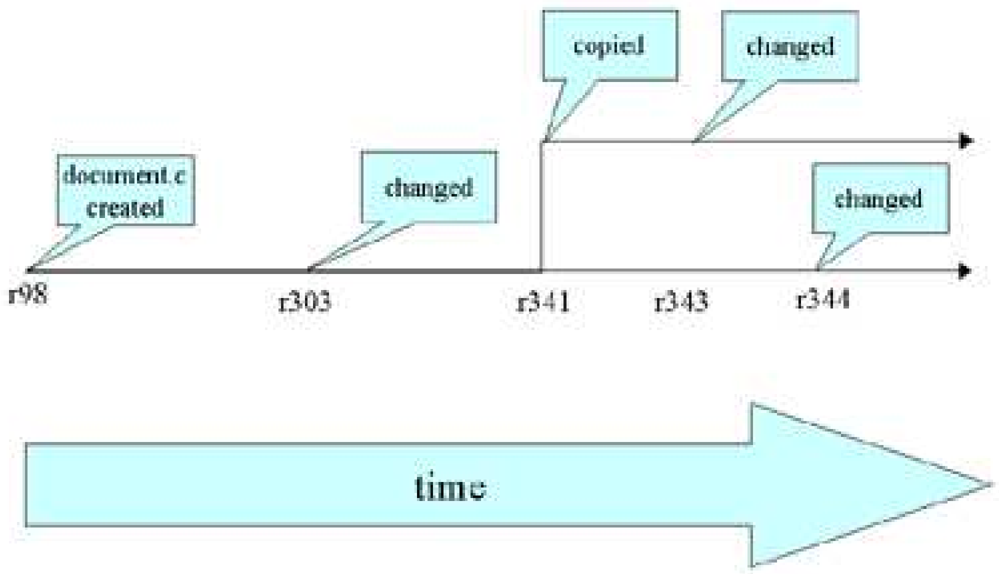
Notice that Subversion is tracing the history of your `document.c` all the way back through time, traversing the point where it was copied. (Remember that your branch was created in revision 341.) Now look what happens when Felix runs the same command on his copy of the file:

```
$ pwd
/home/felix/write

$ svn log document.c
------------------------------------------------------------------------
rev 344:  felix | 2002-11-07 15:27:56 -0600 (Thu, 07 Nov 2002) | 2 lines

* document.c:  fix a bunch of spelling errors.

------------------------------------------------------------------------
rev 303:  felix | 2002-10-29 21:14:35 -0600 (Tue, 29 Oct 2002) | 2 lines

* document.c:  changed a docstring.

------------------------------------------------------------------------
rev 98:  felix | 2002-02-22 15:35:29 -0600 (Fri, 22 Feb 2002) | 2 lines

* document.c:  adding this file to the project.

------------------------------------------------------------------------
```

Felix sees his own revision 344 change, but not the change you made in revision 343. As far as Subversion is concerned, these two commits affected different files in different repository locations. However, Subversion *does* show that the two files share a common history. Before the branch-copy was made in revision 341, they used to be the same file. That's why you and Felix both see revisions 303 and 98.

There are now two independent lines of development happening. A diagram makes it easier to visualize:

Figure 4.4. The branching of one file's history



The moral of the story

There are two important lessons that you should remember from this section.

1. Unlike many other version control systems, Subversion's branches exist in *normal filesystem space*, not in some imaginary extra dimension.

2. Subversion has no internal concept of a "branch" —only copies. When you copy a directory, the resulting directory is only a "branch" because *you* attach that meaning to it. You may think of the directory differently, or treat it differently, but to Subversion it's just an ordinary directory that happens to have been created by copying.

Copying changes between branches

Okay, so now you and Felix are working on parallel branches of the project. The good news is that you're not interfering with each other. The bad news is that it's very easy to drift *too* far apart. Remember that one of the problems with the "crawl in a hole" strategy is that by the time you're finished with your branch, it may be near-impossible to merge your changes back into the main branch.

Instead, you and Felix should continue to share changes as you work. It's up to you to decide which changes are worth sharing; Subversion can gives you the ability to selectively "copy" changes between branches. Here's how.

In the previous section, we mentioned that both you and Felix made changes to `document.c` on different branches. If you look at Felix's log message for revision 344, you can see that he fixed some spelling errors. No doubt, your copy of the same file still has the same spelling errors. It's likely that your future changes to this file will be affecting the same areas that have the spelling errors, so you're in for some potential conflicts when you merge your branches someday. It's better, then, to receive Felix's change now, *before* you start working too heavily in the same places.

Enter the **svn merge** command. This command, it turns out, is a very close cousin to the **svn diff** command (which you read about in Chapter 3.) Both commands are able to compare any two objects in the repository and describe the differences. For example, you can ask **svn diff** to show you the exact change made by Felix in revision 344:

```
$ svn diff -r343:344 http://svn.example.com/repos/trunk/write

Index: document.c
===================================================================
--- document.c (revision 343)
+++ document.c (revision 344)
@@ -147,7 +147,7 @@
     case 6:  sprintf(info->operating_system, "HPFS (OS/2 or NT)"); break;
     case 7:  sprintf(info->operating_system, "Macintosh"); break;
     case 8:  sprintf(info->operating_system, "Z-System"); break;
-    case 9:  sprintf(info->operating_system, "CPM"); break;
+    case 9:  sprintf(info->operating_system, "CP/M"); break;
     case 10:  sprintf(info->operating_system, "TOPS-20"); break;
     case 11:  sprintf(info->operating_system, "NTFS (Windows NT)"); break;
     case 12:  sprintf(info->operating_system, "QDOS"); break;
@@ -164,7 +164,7 @@
     low = (unsigned short) read_byte(gzfile);  /* read LSB */
     high = (unsigned short) read_byte(gzfile); /* read MSB */
     high = high << 8;  /* interpret MSB correctly */
-    total = low + high; /* add them togethe for correct total */
+    total = low + high; /* add them together for correct total */

     info->extra_header = (unsigned char *) my_malloc(total);
     fread(info->extra_header, total, 1, gzfile);
@@ -241,7 +241,7 @@
      Store the offset with ftell() ! */

   if ((info->data_offset = ftell(gzfile))== -1) {
-    printf("error: ftell() retturned -1.\n");
+    printf("error: ftell() returned -1.\n");
     exit(1);
   }

@@ -249,7 +249,7 @@
   printf("I believe start of compressed data is %u\n", info->data_offset);
   #endif

-  /* Set postion eight bytes from the end of the file. */
+  /* Set position eight bytes from the end of the file. */

   if (fseek(gzfile, -8, SEEK_END)) {
     printf("error: fseek() returned non-zero\n");
```

The **svn merge** is almost exactly the same. Instead of printing the differences to your terminal, however, it applies them directly to your working copy as *local modifications*:

```
$ svn merge -r343:344 http://svn.example.com/repos/trunk/write
U  document.c

$ svn status
M  document.c
```

Your working copy now contains Felix's change ˙ it has been "copied" from his branch to your working copy. At this point, it's up to you to review the local modifications and make sure they're all good. In another scenario, it's possible that things may not have gone so well, and that `document.c` may have entered a conflicted state. You might need to resolve the conflict by hand, or if you're really disgusted, simply give up and **svn revert** the local change. But assuming the changes are working and you're confident that the merge was good, you can **svn commit** the change as usual. At that point, the change has been merged into your repository branch. In version control terminology, this act of copying changes is commonly called porting changes between branches.

A question may be on your mind, especially if you're a Unix user: why bother to use **svn merge** at all? Why not simply use the operating system's **patch** command to accomplish the same job? For example:

```
$ svn diff -r343:344 http://svn.example.com/repos/trunk/write > patchfile
$ patch -p0  < patchfile
Patching file document.c using Plan A...
Hunk #1 succeeded at 147.
Hunk #2 succeeded at 164.
Hunk #3 succeeded at 241.
Hunk #4 succeeded at 249.
done
```

In this particular case, yes, there really is no difference. But **svn merge** has special abilities that surpass the **patch** program. The file format used by **patch** is quite limited; it's only able to tweak file contents. There's no way to represent changes to *trees*, such as the addition, removal, or renaming of files and directories. If Felix's change had, say, added a new directory, the output of **svn diff** wouldn't have mentioned it at all. **svn diff** only outputs the limited patch-format, so there are some ideas it simply can't express. The **svn merge** command, however, can express tree-changes by directly applying them to your working copy.

A word of warning: while **svn diff** and **svn merge** are very similar in concept, they do have different syntax in many cases. Be sure to read about them in Chapter 8 for details, or ask **svn help**. For example, **svn merge** insists on a working-copy path as a 'target', i.e. a place where it should apply the tree-changes. It defaults to the current working directory, so if you want changes applied somewhere else, you'll need to say so:

```
$ svn merge -r343:344 http://svn.example.com/repos/trunk/write my-write-branch
U   my-write-branch/document.c
```

The repeated merge problem

Merging changes from one branch to another sounds simple enough, but in practice it can sometimes become a headache. The problem is that if you repeatedly merge changes from branch A to branch B, you may accidentally merge the same change *twice*. When this happens, you'll often get a conflict: Subversion will attempt to contextually merge a change into a file that already contains the change.

This is a problem that plagues many version control systems, including both CVS and Subversion. For now, the only way to avoid this problem in Subversion is to carefully keep track of which changes have been merged, and which haven't. Specifically, when you create a branch directory, remember what revision it was created in. When you merge a revision (or range of revisions) into your working copy, you'll need to remember them as well. If you forget any of this information, you can rediscover it by examining the output of **svn log --verbose branch-dir**. But the point is that each subsequent merge needs to be carefully constructed by hand, making sure that previously-merged revisions aren't re-merged again.

Of course, Subversion has plans to solve this problem sometime after release 1.0. All of this merging information can be tracked in property metadata (see the section called ˆProperties˜), and thus Subversion will someday be able to automatically avoid repeated merges.

Removing a change from the repository

A common use for **svn merge** is to roll back a change that has already been committed. Suppose you decide that the change made way back in revision 303, which changed a docstring in `document.c`, is completely wrong. It never should have been committed. You can use **svn merge** to "undo" the change in your working copy, and then commit the local modification to the repository. All you need to do is do specify a *reverse* difference:

```
$ svn merge -r 303:302 http://svn.example.com/trunk/write
U  document.c

$ svn status
M  document.c

$ svn commit -m "Undoing change committed in r303."
Sending    document.c
Transmitting file data .
Committed revision 350.
```

One way to think about repository revisions is as changesets. That is, each revision number is the name of a particular group of changes. By using the `-r` switch, you can ask **svn merge** to apply a changeset, or whole range of changesets, to your working copy. In our case of undoing a change, we're asking **svn merge** to apply changeset #303 to our working copy *backwards*.

Keep in mind that rolling back a change like this is just like any other **svn merge** operation, so you should use **svn status** and **svn diff** to confirm that your work is in the state you want it to be in, and then use **svn commit** to send the final version to the repository. After committing, this particular changeset is no longer reflected in the HEAD revision.

Again, you may be thinking: well, that really didn't undo the commit, did it? The change still exists in revision 303. If somebody checks out a version of the `write` project between revisions 303 and 349, they'll still see the bad change, right?

Yes, that's true. When we talk about "removing" a change, we're really talking about "removing it from HEAD". The original change still exists in the repository's history. For most situations, this is good enough. Most people are only interested in tracking the HEAD of a project anyway. There are special cases, however, where you really might want to destroy all evidence of the commit. (Perhaps somebody accidentally committed a confidential document.) This isn't so easy, it turns out, because Subversion was deliberately designed to never lose information. Revisions are assumed to be immutable, and thus build upon one another. Removing a revision from history would cause a domino effect, creating chaos in all subsequent revisions and possibly invalidating all working copies.

The Subversion project has plans, however, to someday implement an **svnadmin obliterate** command. It would probably involve taking the repository off-line, removing a revision, and then "rewriting" all subsequent revisions. At the time of writing, however, this command isn't scheduled to exist until sometime after the release of Subversion 1.0.

Tags

Another common version control concept is a tag. A tag is just a "snapshot" of a project in time. In Subversion, this idea already seems to be everywhere. Each repository revision is exactly that ˙ a snapshot of the filesystem after each commit.

However, people often want to give more human-friendly names to tags, like "beta-5". And they want to make snapshots of smaller subdirectories of the filesystem. After all, it's not so easy to remember that the fifth beta candidate of a piece of software is a particular subdirectory of revision 4822.

Once again, **svn copy** comes to the rescue. What could be easier? If you want to create a snapshot of `/trunk/write` exactly as it looks in the HEAD revision, then make a copy of it:

```
$ svn copy http://svn.example.com/repos/trunk/write \
           http://svn.example.com/repos/tags/write/beta-5 \
      -m "Tagging the beta-5 release of the 'write' project."

Committed revision 351.
```

This example assumes that a `/tags/write` directory already exists. After the copy completes, the new `beta-5` directory is forever a snapshot of how the project looked in the HEAD revision at the time you made the copy. Of course you might want to be more precise about exactly which revision you copy, in case somebody else may have committed changes to the project when you weren't looking. So if you know that revision 350 of `/trunk/write` is exactly the snapshot you want, you can specify it by passing the `-r350` option to the copy command.

But wait a moment: isn't this tag-creation procedure the same procedure we used to create a branch? Yes, in fact, it is. The shocking truth is that in Subversion, there's no difference between a tag and a branch. Tags and branches are both just ordinary directories that are created by copying. As we mentioned earlier, the only reason a copied directory is a "tag" is because *humans* have decided to treat it that way: as long as nobody ever commits to the directory, it forever remains a snapshot. If people start committing to it, it becomes a branch.

If you are administering a repository, there are two approaches you can take to managing tags. The first approach is "hands off": as a matter of project policy, decide where your tags will live, and make sure all users know how to treat the directories they copy in there. (That is, make sure they know not to commit to them.) The second approach is more paranoid: you can use one of the access-control scripts provided with Subversion to prevent anyone from doing anything but creating new copies in the tags-area (###cross-ref a section that demonstrates how to use these scripts?). The paranoid approach, however, isn't usually necessary. If a user accidentally commits a change to a tag-directory, you can simply undo the change as discussed in the previous section. This is version control, after all!

Branch maintenance

You may have noticed by now that Subversion is extremely flexible. Because it implements branches and tags with the same underlying mechanism (directory copies), and because branches and tags appear in normal filesystem space, many people find Subversion intimidating. It's almost *too* flexible.

For this reason, there are some standard, recommended ways to lay out a repository. Most people create a `trunk` directory to hold the "main line" of development, a `branches` directory to contain branch copies, and a `tags` directory to contain tag copies. If a repository holds only one project, then often people create these top-level directories:

```
/trunk
/branches
/tags
```

If a repository contains multiple projects, people often index their layout by branch:

```
/trunk/paint
/trunk/write
/branches/paint
/branches/write
/tags/paint
/tags/write
```

...or by project:

```
/paint/trunk
/paint/branches
/paint/tags
/write/trunk
/write/branches
/write/tags
```

Of course, you're free to ignore these common layouts. You can create any sort of variation, whatever works best for you or your team. Remember that whatever you choose, it's not a permanent commitment. You can reorganize your repository at any time. Because branches and tags are ordinary directories, the **svn mv** command can move or rename them however you wish. Switching from one layout to another is just a matter by issuing a series of server-side moves; if you don't like the way things are organized in the repository, just juggle the directories around!

Another nice feature of Subversion's model is that branches and tags can have finite lifetimes, just like any other versioned item. For example, suppose you eventually finish all your work on your personal branch of the `write` project. After merging all of your changes back into `/trunk/write`, there's no need for your private branch to stick around anymore:

```
$ svn rm http://svn.example.com/repos/branches/write/my-write-branch \
         -m "Removing obsolete branch of write project."

Committed revision 375.
```

And now your branch is gone. Of course it's not really gone: the directory is simply missing from the HEAD revision, no longer distracting anyone. If you look at an earlier revision, of course, you'll still be able to find your old branch.

In this case, your branch had a relatively short lifetime: you may have created it to fix a bug or implement a new feature. When your task is done, so is the branch. In software development, though, it's also common to have two "main branches running side-by-sidefor very long periods. For example, suppose it's time to release a stable `write` project to the public, and you know it's going to take a couple of months to shake bugs out of the software. You don't want people to add new features to the project, but you don't want to tell all developers to stop programming either. So instead, you create a "stable" branch of the software that won't change much:

```
$ svn cp http://svn.example.com/repos/trunk/write \
         http://svn.example.com/branches/write/release-1.0
         -m "Creating stable release branch of write project."

Committed revision 377.
```

And now developers are free to continue adding cutting-edge(or experimental) features to `/trunk/write`, and you can declare a project policy that only bugfixes are to be committed to `/branches/write/release-1.0`. That is, as people continue to work on the trunk, a human selectively ports bugfixes over to the release branch. Even after the release branch has shipped out the door, you'll probably continue to maintain the branch for a long time ˙ that is, as long as you continue to support that release for customers.

Switching a working copy

Subversion has one final trick up its sleeve: the **svn switch** command.

The **svn switch** transforms an existing working copy into a different branch. While this command isn't strictly necessary for working with branches, it provides a nice shortcut to users. In our earlier example, after creating your private branch, you checked out a fresh working copy of the new repository directory. Instead, you could simply ask Subversion to change your working copy of `/trunk/write` to mirror the new branch location:

```
$ cd write

$ svn info | grep Url
Url: http://svn.example.com/trunk/write

$ svn switch http://svn.example.com/branches/write/my-write-branch
U   document.c
U   search.c
U   Makefile
Updated to revision 341.

$ svn info | grep Url
Url: http://svn.example.com/branches/write/my-write-branch
```

After "switching" to the branch, your working copy is no different than what you would get from doing a fresh checkout of the directory. And it's usually more efficient to use this command, because often branches only differ by a small degree. The server sends only the minimal set of changes necessary to make your working copy reflect the branch.

Of course, most projects are more complicated than our `write` example. Users often create a branch of a project's subdirectory, and then use **svn switch** to move that specific subdirectory in their working copy to the new branch. Or sometimes users will only switch a single working file to a branch. That way, they can continue to receive normal updates to most of their working copy, but the "switched" portions will remain immune (unless someone commits a change to their branch.) This feature adds a whole new dimension to the concept of a "mixed working copy": not only can working copies contain a mixture of working revisions, but a mixture of repository locations as well.

If your working copy contains a number of switched subtrees from different repository locations, it continues to function as normal. When you update, you'll receive patches to each subtree as appropriate. When you commit, your local changes will still be applied as a single, atomic change to the repository.

A final note: while it's okay for your working copy to reflect a hodgepodge of repository locations, these locations must all be within the *same* repository. Subversion repositories aren't yet able to communicate with one another; that's a feature planned beyond Subversion 1.0.

Switches and updates

Have you noticed that the output of **svn switch** and **svn update** look the same? That's because they *are* the same.

The switch command is actually a strict superset of the update command. For people curious about implementation, the commands work like this:

- The client describes the working copy to the repository, mixed revisions and all.
- A temporary tree is constructed within the repository that is an exact replica of the working copy.
- The repository compares the temporary tree with some revision tree, and sends a tree–delta back to the client.
- The client applies the tree–delta to the working copy, so the working copy now reflects the revision tree.

The only difference between **svn switch** and **svn update** is that the update command compares two identical repository locations. That is, if your working copy is a mirror of `/trunk/write`, then **svn update** will automatically compare your working copy to `/trunk/write` in the HEAD revision. If you're switching your working copy to a branch, then **svn switch** will compare your working copy to some other branch–directory in the HEAD revision.

In other words, an update moves your working copy through time. A switch moves your working copy through time *and* space.

Summary

We've covered a lot of ground in this chapter. We've discussed the concepts of tags and branches, and demonstrated how Subversion implements these concepts by copying directories with the **svn copy** command. We've shown how to use **svn merge** to copy changes from one branch to another, or roll back bad changes. We've gone over the use of **svn switch** to create mixed–location working copies. And we've talked about how one might manage the organization and lifetimes of branches in a repository.

Remember the Subversion mantra: branches are cheap. So use them liberally!

Chapter 5. Repository Administration

The Subversion repository is the central storehouse of versioned data for any number of projects. As such, it becomes an obvious candidate for all the love and attention an administrator can offer. While the repository is generally a low−maintenanceitem, it is important to understand how to properly configure and care for it so that potential problems are avoided, and actual problems safely resolved.

In this chapter, we'll discuss how to create and configure a Subversion repository, and how to expose that repository for network accessibility. We'll also talk about repository maintenance, including the use of the **svnlook** and **svnadmin** tools (which are provided with Subversion). We'll address some common questions and mistakes, and give some suggestions on how to arrange the data in the repository.

If you plan to access a Subversion repository only in the role of a user whose data is under version control (that is, via a Subversion client), you can skip this chapter altogether. However, if you are, or wish to become, a Subversion repository administrator, 2 you should definitely pay attention to this chapter.

Of course, one cannot administer a repository unless one has a repository to administer.

Repository Basics

Understanding Transactions and Revisions

Conceptually speaking, a Subversion repository is a sequence of directory trees. Each tree is a snapshot of how the files and directories versioned in your repository looked at various points in time. These snapshots are created as a result of client operations, and are called revisions.

Every revision begins life as a transaction tree. When doing a commit, a client builds a Subversion transaction that mirrors their local changes (plus any additional changes that might have been made to the repository since the beginning of the client's commit process), and then instructs the repository to store that tree as the next snapshot in the sequence. If the commit succeeds, the transaction is effectively promoted into a new revision tree, and is assigned a new revision number. If the commit fails for some reason, the transaction is destroyed and the client is informed of the failure.

At the moment, updates work in a similar way. The client builds a temporary transaction tree that mirrors the state of the working copy. The repository then compares that transaction tree with the revision tree at the request revision (usually the most recent, or ˆyoungest˜ tree), and sends back information that informs the client about what changes are needed to transform their working copy into a replica of that revision tree. After the update completes, the temporary transaction is deleted.

The use of transaction trees is the only way to make permanent changes to repository's versioned filesystem. However, it's important to understand that the lifetime of a transaction is completely flexible. In the case of updates, transactions are temporary trees that are immediately destroyed. In the case of commits, transactions are transformed into permanent revisions (or removed if the commit fails). In the case of an error or bug, it's possible that a transaction can be accidentally left lying around in the repository (not really affecting anything, but still taking up space).

In theory, someday whole workflow applications might revolve around more fine−grainedcontrol of transaction lifetime. It is feasible to imagine a system whereby each transaction slated to become a revision is left in stasis well after the client finishes describing its changes to repository. This would enable each new commit to be reviewed by someone else, perhaps a manager or engineering QA team, who can choose to promote the transaction into a revision, or abort it.

What does all of this have to do with repository administration? The answer is simple: if you're administering a Subversion repository, you're going to have to examine revisions and transactions as part of monitoring the health of your repository.

2This may sound really prestigious and lofty, but we're just talking about anyone who is interested in that mysterious realm beyond the working copy where everyone's data hangs out.

Unversioned Properties

Transactions and revisions in the Subversion repository can have properties attached to them. These properties are generic key−to−value mappings, and are used to store information about the tree to which they are attached. The names and values of these properties are stored in the repository's filesystem, along with the rest of your tree data.

Revision and transaction properties are useful for associating information with a tree that is not strictly related to the files and directories in that tree˙ the kind of information that isn't managed by client working copies. For example, when a new commit transaction is created in the repository, Subversion adds a property to that transaction named `svn:date`˙ a datestamp representing the time that the transaction was created. By the time the commit process is finished, and the transaction is promoted to a permanent revision, the tree has also been given a property to store the username of the revision's author (`svn:author`) and a property to store the log message attached to that revision (`svn:log`).

Revision and transaction properties are unversioned properties˙ as they are modified, their previous values are permanently discarded. Also, while revision trees themselves are immutable, the properties attached to those trees are not. You can add, remove, and modify revision properties at any time in the future. If you commit a new revision and later realize that you had some misinformation or spelling error in your log message, you can simply replace the value of the `svn:log` property with a new, corrected log message.

Repository Creation and Configuration

Creating a Subversion repository is an incredibly simple task. The **svnadmin** utility, provided with Subversion, has a subcommand for doing just that. To create a new repository, just run:

```
$ svnadmin create path/to/repos
```

This creates a new repository in the directory `path/to/repos`. This new repository begins life at revision 0, which is defined to consist of nothing but the top−level root (`/`) filesystem directory. Initially, revision 0 also has a single revision property, `svn:date`, set to the time at which the repository was created.

You may have noticed that the path argument to **svnadmin** was just a regular filesystem path and not a URL like the **svn** client program uses when referring to repositories. Both **svnadmin** and **svnlook** are considered server−side utilities˙ they are used on the machine where the repository resides to examine or modify aspects of the repository, and are in fact unable to perform tasks across a network. A common mistake made by Subversion newcomers is trying to pass URLs (even "local" `file:` ones) to these two programs.

So, after you've run the **svnadmin create** command, you have a shiny new Subversion repository in its own directory. Let's take a peek at what is actually created inside that subdirectory.

```
$ ls repos
conf/   dav/   db/   format   hooks/   locks/   README
```

With the exception of the `README` file, the repository directory is a collection of subdirectories. As in other areas of the Subversion design, modularity is given high regard, and hierarchical organization is preferred to cluttered chaos. Here is a brief description of all of the items you see in your new repository directory:

conf
: Currently unused, repository configuration files will go in this directory someday.

dav
: A directory provided to Apache and mod_dav_svn for their private housekeeping data.

db
: The main Berkeley DB environment, full of DB tables that comprise the data store for Subversion's filesystem (where all of your versioned data resides).

format
: A file whose contents are a single integer value that dictates the version number of the repository layout.

hooks
: A directory full of hook script templates (and hook scripts themselves, once you've installed some).

locks A directory for Subversion's repository locking data, used for tracking accessors to the repository.

README A file which merely informs its readers that they are looking at a Subversion repository.

In general, you shouldn't tamper with your repository "by hand". The **svnadmin** tool should be sufficient for any changes necessary to your repository, or you can look to third–party tools (such as Berkeley DB's tool suite) for tweaking relevant subsections of the repository. Some exceptions exist, though, and we'll cover those here.

Hook scripts

A hook is a program triggered by some repository event, such as the creation of a new revision or the modification of an unversioned property. Each hook is handed enough information to tell what that event is, what target(s) it's operating on, and the username of the person who triggered the event. Depending on the hook's output or return status, the hook program may continue the action, stop it, or suspend it in some way.

The `hooks` subdirectory is, by default, filled with templates for various repository hooks.

```
$ ls repos/hooks/
post-commit.tmpl          pre-revprop-change.tmpl  write-sentinels.tmpl
post-revprop-change.tmpl  read-sentinels.tmpl
pre-commit.tmpl           start-commit.tmpl
```

There is one template for each hook that the Subversion repository implements, and by examining the contents of those template scripts, you can see what triggers each such script to run and what data is passed to that script. Also present in many of these templates are examples of how one might use that script, in conjunction with other Subversion–supplied programs, to perform common useful tasks. To actually install a working hook, you need only place some executable program or script into the `repos/hooks` directory which can be executed as the name (like **start–commit** or **post–commit**) of the hook.

On Unix platforms, this means supplying a script or program (which could be a shell script, a Python program, a compiled C binary, or any number of other things) named exactly like the name of the hook. Of course, the template files are present for more than just informational purposes— the easiest way to install a hook on Unix platforms is to simply copy the appropriate template file to a new file that lacks the `.tmpl` extension, customize the hook's contents, and ensure that the script is executable. Windows, however, uses file extensions to determine whether or not a program is executable, so you would need to supply a program whose basename is the name of the hook, and whose extension is one of the special extensions recognized by Windows for executable programs, such as `.exe` or `.com` for programs, and `.bat` for batch files.

Currently there are five true hooks implemented by the Subversion repository.

`start-commit` This is run before the commit transaction is even created. It is typically used to decide if the user has commit privileges at all. The repository passes two arguments to this program: the path to the repository, and username which is attempting the commit. If the program returns a non–zero exit value, the commit is stopped before the transaction is even created.

`pre-commit` This is run when the transaction is complete, but before it is committed. Typically, this hook is used to protect against commits that are disallowed due to content or location (for example, your site might require that all commits to a certain branch include a ticket number from the bug tracker, or that the incoming log message is non–empty). The repository passes two arguments to this program: the path to the repository, and the name of the transaction being committed. If the program returns a non–zero exit value, the commit is aborted and transaction is removed.

The Subversion distribution includes some access control scripts (located in the `tools/hook-scripts` directory of the Subversion source tree) that can be called from **pre–commit** to implement fine–grained access control. At this time, this is the only method by which administrators can implement finer–grained access control beyond what `httpd.conf` offers. In a future version of Subversion, we plan to implement ACLs directly in the filesystem.

`post-commit`
: This is run after the transaction is committed, and a new revision is created. Most people use this hook to send out descriptive emails about the commit or to make a backup of the repository. The repository passes two arguments to this program: the path to the repository, and the new revision number that was created. The exit code of the program is ignored.

  The Subversion distribution includes a **commit–email.pl** script (located in the `tools/hook-scripts/` directory of the Subversion source tree) that can be used to send email with (and/or append to a log file) a description of a given commit. This mail contains a list of the paths that were changed, the log message attached to the commit, the author and date of the commit, as well as a GNU diff–style display of the changes made to the various versioned files as part of the commit.

  Another useful tool provided by Subversion is the **hot–backup.py** script (located in the `tools/backup/` directory of the Subversion source tree). This script performs hot backups of your Subversion repository (a feature supported by the Berkeley DB database back–end), and can be used to make a per–commit snapshot of your repository for archival or emergency recovery purposes.

`pre-revprop-change`
: Because Subversion's revision properties are not versioned, making modifications to such a property (for example, the `svn:log` commit message property) will overwrite the previous value of that property forever. Since data can be potentially lost here, Subversion supplies this hook (and its counterpart, `post-revprop-change`) so that repository administrators can keep records of changes to these items using some external means if they so desire.

  This hook runs just before such a modification is made to the repository. The repository passes four arguments to this hook: the path to the repository, the revision on which the to–be–modified property exists, the authenticated username of the person making the change, and name of the property itself.

`post-revprop-change`
: As mentioned earlier, this hook is the counterpart of `pre-revprop-change` hook. In fact, for the sake of paranoia this script will not run unless the `pre-revprop-change` hook exists. When both of these hooks are present, the `post-revprop-change` hook runs just after a revision property has been changed, and is typically used to send an email containing the new value of the changed property. The repository passes four arguments to this hook: the path to the repository, the revision on which the property exists, the authenticated username of the person making the change, and name of the property itself.

  The Subversion distribution includes a **propchange–email.pl** script (located in the `tools/hook-scripts/` directory of the Subversion source tree) that can be used to send email with (and/or append to a log file) the details of a revision property change. This mail contains the revision and name of the changed property, the user who made the change, and the new property value.

Sentinels, Those ˆOther˜ Hooks

In addition to the hook scripts, Subversion has templates for two other special event handlers˙ the `read-sentinels` and `write-sentinels`. These special programs, called sentinels, are intended to be daemon–like applications which are started at the beginning of a user's operation, and which receive multiple event notifications of a

> single type over the course of that operation. Depending on the sentinel's responses to each of these notifications, Subversion may stop or otherwise modify the operation. Fortunately for today's Subversion repository administrators, these sentinels can be ignored, since support for them is not yet implemented in Subversion itself!

Subversion will attempt to execute hooks as the same user who owns the process which is accessing the Subversion repository. In most cases, the repository is being accessed via Apache HTTP server and mod_dav_svn, so this user is the same user that Apache runs as. The hooks themselves will need to be configured with OS−level permissions that allow that user to execute them. Also, this means that any file or programs (including the Subversion repository itself) accessed directly or indirectly by the hook will be accessed as the same user. In other words, be alert to potential permission−related problems that could prevent the hook from performing the tasks you've written it to perform.

Berkeley DB configuration

The Berkeley DB environment has it own set of default configuration values for things like the number of locks allowed to be taken out at any given time, or the size cutoff for Berkeley's journaling log file, etc. Subversion's filesystem code additionally chooses default values for some of the Berkeley DB configuration options. However, sometimes your particular repository, with its unique collection of data and access patterns, might require a different set of configuration option values.

The folks at Sleepycat (the producers of Berkeley DB) understand that different databases have different requirements, and so they have provided a mechanism for runtime overriding of many of the configuration values for the Berkeley DB environment. Berkeley checks for the presence of a file named `DB_CONFIG` in each environment directory, and parses the options found in that file for use with that particular Berkeley environment.

The Berkeley configuration file for your repository is located in the `db` environment directory, at `repos/db/DB_CONFIG`. Subversion itself creates this file when it creates the rest of the repository. The file initially contains some default options, as well as pointers to the Berkeley DB online documentation so you can read about what those options do. Of course, you are free to add any of the supported Berkeley DB options to your `DB_CONFIG` file. Just be aware that while Subversion never attempts to read or interpret the contents of the file, and makes no use of the option settings in it, you'll want to avoid any configuration changes that may cause Berkeley DB to behave in a fashion that is unexpected by the rest of the Subversion code.

Repository Maintenance

An Administrator's Toolkit

svnlook

svnlook is a tool provided by Subversion for examining the various revisions and transactions in a repository. No part of this program attempts to change the repository˙ it's a ˆread−only˜ tool. **svnlook** is typically used by the repository hooks for reporting the changes that are about to be committed (in the case of the **pre−commit** hook) or that were just committed (in the case of the **post−commit** hook) to the repository. A repository administrator may use this tool for diagnostic purposes.

svnlook has a straightforward syntax:

```
$ svnlook help
general usage: svnlook SUBCOMMAND REPOS_PATH [ARGS & OPTIONS ...]
Note: any subcommand which takes the '--revision' and '--transaction'
      options will, if invoked without one of those options, act on
      the repository's youngest revision.
Type "svnlook help <subcommand>" for help on a specific subcommand.
```

Nearly every one of **svnlook**'s subcommands can operate on either a revision or a transaction tree, printing information about the tree itself, or how it differs from the previous revision of the repository. You use the `--revision` and

`--transaction` options to specify which revision or transaction, respectively, to examine. Note that while revision numbers appear as natural numbers, transaction names are alphanumeric strings. Keep in mind that the filesystem only allows browsing of uncommitted transactions (transactions that have not resulted in a new revision). Most repositories will have no such transactions, because transactions are usually either committed (which disqualifies them from viewing) or aborted and removed.

In the absence of both the `--revision` and `--transaction` options, **svnlook** will examine the youngest (or ˆHEAD˜) revision in the repository. So the following two commands do exactly the same thing when 19 is the youngest revision in the repository located at `/path/to/repos`:

```
$ svnlook info /path/to/repos
$ svnlook info /path/to/repos --revision 19
```

The only exception to these rules about subcommands is the **svnlook youngest** subcommand, which takes no op‒tions, and simply prints out the HEAD revision number.

```
$ svnlook youngest /path/to/repos
19
```

Output from **svnlook** is designed to be both human‒ and machine‒parsable. Take as an example the output of the `info` subcommand:

```
$ svnlook info path/to/repos
sally
2002-11-04 09:29:13 -0600 (Mon, 04 Nov 2002)
27
Added the usual
Greek tree.
```

The output of the `info` subcommand is defined as:

1. The author, followed by a newline.
2. The date, followed by a newline.
3. The number of characters in the log message, followed by a newline.
4. The log message itself, followed by a newline.

This output is human‒readable,meaning items like the datestamp are displayed using a textual representation instead of something more obscure (such as the number of nanoseconds since the Tasty Freeze guy drove by). But this out‒put is also machine‒parsable˙ because the log message can contain multiple lines and be unbounded in length, **svn‒look** provides the length of that message before the message itself. This allows scripts and other wrappers around this command to make intelligent decisions about the log message, such as how much memory to allocate for the message, or at least how many bytes to skip in the event that this output is not the last bit of data in the stream.

Another common use of **svnlook** is to actually view the contents of a revision or transaction tree. Examining the out‒put of **svnlook tree** command, which displays the directories and files in the requested tree (optionally showing the filesystem node revision IDs for each of those paths) can be extremely helpful to administrators deciding on whether or not it is safe to remove a seemingly dead transaction. It is also quite useful for Subversion developers who are di‒agnosing filesystem‒relatedproblems when they arise.

```
$ svnlook tree path/to/repos --show-ids
/ <0.0.1>
 A/ <2.0.1>
  B/ <4.0.1>
   lambda <5.0.1>
   E/ <6.0.1>
```

```
    alpha <7.0.1>
    beta <8.0.1>
   F/ <9.0.1>
  mu <3.0.1>
  C/ <a.0.1>
  D/ <b.0.1>
   gamma <c.0.1>
   G/ <d.0.1>
    pi <e.0.1>
    rho <f.0.1>
    tau <g.0.1>
   H/ <h.0.1>
    chi <i.0.1>
    omega <k.0.1>
    psi <j.0.1>
 iota <1.0.1>
```

svnlook can perform a variety of other queries, displaying subsets of bits of information we've mentioned previously, reporting which paths were modified in a given revision or transaction, showing textual and property differences made to files and directories, and so on. The following is a brief description of the current list of subcommands accepted by **svnlook**, and the output of those subcommands:

`author` Print the tree's author.

`date` Print the tree's datestamp.

`changed` List all files and directories that changed in the tree.

`diff` Print unified diffs of changed files.

`dirs-changed` List the directories that changed in the tree.

`info` Print the tree's author, datestamp, log message character count, and log message.

`log` Print the tree's log message.

`tree` Print the tree listing, optionally revealing the filesystem node revision IDs associated with each path.

`youngest` Print the youngest revision number.

svnadmin

The **svnadmin** program is the repository administrator's best friend. Besides providing the ability to create Subversion repositories, this program allows you perform several maintenance operations on those repositories. The syntax of **svnadmin** is similar to that of **svnlook**:

```
$ svnadmin help
general usage: svnadmin SUBCOMMAND REPOS_PATH  [ARGS & OPTIONS ...]
Type "svnadmin help <subcommand>" for help on a specific subcommand.

Available subcommands:
   create
   createtxn
   dump
```

We've already mentioned **svnadmin**'s `create` subcommand (see the section called ˆRepository Creation and Configuration˜). Most of the others we will cover in more detail later in this chapter. For now, let's just take a quick glance at what each of the available subcommands offers.

`create` Creates a new Subversion repository.

`createtxn` Creates a transaction in the repository based on a given existing revision.

`dump` Dumps the contents of the repository, bounded by a given set of revisions, using a portable dump format.

`load` Loads a set of revisions into a repository from a stream of data that uses the same portable dump format generated by the `dump` subcommand.

`lscr` List the revisions in which a given path in the repository was modified.

`lstxns` List the names of uncommitted Subversion transactions which currently exist in the repository.

`recover` Perform recovery steps on a repository that is in need of such, generally after a fatal error has occurred which prevented a process from cleanly shutting down its communication with the repository.

`rmtxns` Cleanly remove Subversion transactions from the repository (conveniently fed by output from the `lstxns` subcommand).

`setlog` Replace the current value of the `svn:log` (commit log message) property on a given revision in the repository with a new value.

svnshell.py

The Subversion source tree also comes with a shell-like interface to the repository. The **svnshell.py** Python script (located in `tools/examples/` in the source tree) uses Subversion's language bindings (so you have to have those properly compiled and installed in order for this script to work) to connect to the repository and filesystem libraries.

Once started, the program behaves similarly to a shell program, allowing you to browse the various directories in your repository. Initially, you are ˆpositioned˜ in the root directory of the HEAD revision of the repository, and presented with a command prompt. You can use the `help` command at any time to display a list of available commands and what they do.

```
$ svnshell.py /path/to/repos
<rev: 2 />$  help
Available commands:
  cat FILE     : dump the contents of FILE
  cd DIR       : change the current working directory to DIR
  exit         : exit the shell
  ls [PATH]    : list the contents of the current directory
  lstxns       : list the transactions available for browsing
  setrev REV   : set the current revision to browse
  settxn TXN   : set the current transaction to browse
  youngest     : list the youngest browsable revision number
<rev: 2 />$
```

Navigating the directory structure of your repository is done in the same way you would navigate a regular Unix or Windows shell˙ using the `cd` command. At all times, the command prompt will show you what revision (prefixed by `rev:`) or transaction (prefixed by `txn:`) you are currently examining, and at what path location in that revision or transaction. You can change your current revision or transaction with the `setrev` and `settxn` commands, respectively. As in a Unix shell, you can use the `ls` command to display the contents of the current directory, and you can use the `cat` command to display the contents of a file.

```
<rev: 2 />$ ls
   REV   AUTHOR  NODE-REV-ID     SIZE         DATE NAME
----------------------------------------------------------------------------
     1    sally <     2.0.1>          Nov 15 11:50 A/
     2    harry <     1.0.2>       56 Nov 19 08:19 iota
<rev: 2 />$ cd A
<rev: 2 /A>$ ls
   REV   AUTHOR  NODE-REV-ID     SIZE         DATE NAME
```

```
----------------------------------------------------------------------------
     1    sally <     4.0.1>          Nov 15 11:50 B/
     1    sally <     a.0.1>          Nov 15 11:50 C/
     1    sally <     b.0.1>          Nov 15 11:50 D/
     1    sally <     3.0.1>       23 Nov 15 11:50 mu
<rev: 2 /A>$ cd D/G
<rev: 2 /A/D/G>$ ls
   REV   AUTHOR  NODE-REV-ID     SIZE         DATE NAME
----------------------------------------------------------------------------
     1    sally <     e.0.1>       23 Nov 15 11:50 pi
     1    sally <     f.0.1>       24 Nov 15 11:50 rho
     1    sally <     g.0.1>       24 Nov 15 11:50 tau
<rev: 2 /A>$ cd ../..
<rev: 2 />$ cat iota
This is the file 'iota'.
Added this text in revision 2.

<rev: 2 />$ setrev 1; cat iota
This is the file 'iota'.

<rev: 1 />$ exit
$
```

As you can see in the previous example, multiple commands may be specified at a single command prompt, separated by a semicolon. Also, the shell understands the notions of relative and absolute paths, and will properly handle the "`.`" and "`..`" special path components.

The `youngest` command displays the youngest revision. This is useful for determining the range of valid revisions you can use as arguments to the `setrev` command˙ you are allowed to browse all the revisions (recalling that they are named with integers) between 0 and the youngest, inclusively. Determining the valid browsable transactions isn't quite as pretty. Use the **lstxns** command to list the transactions that you are able to browse. The list of browsable transactions is the same list that **svnadmin lstxns** returns, and the same list that is valid for use with **svnlook**'s `--transaction` option.

Once you've finished using the shell, you can exit cleanly by using the **exit** command. Alternatively, you can supply an end−of−file character˙ Control−D (though some Win32 Python distributions use the Windows Control−Z convention instead).

Berkeley DB Utilities

Currently, the Subversion repository has only one database back−end˙ Berkeley DB. All of your filesystem's structure and data live in a set of tables within the `db` subdirectory of your repository. This subdirectory is a regular Berkeley DB environment directory, and can therefore be used in conjunction with any of Berkeley's database tools (you can see the documentation for these tools at SleepyCat's website, `http://www.sleepycat.com/`). For day−to−day Subversion use, these tools are unnecessary, however, they do provide some important functionality that is currently not provided by Subversion itself.

For example, because Subversion uses Berkeley DB's logging facilities, the database first writes out a description of any modifications it is about to make, and then makes the modification itself. This is to ensure that if something goes wrong, the database system can back up to a previous checkpoint˙ a location in the log files known not to be corrupt˙ and replay transactions until the data is restored to a usable state. This functionality is one of the main reasons why Berkeley DB was chosen as Subversion's initial database back−end.

Over time, these log files can accumulate. That is actually a feature of the database system˙ you should be able to recreate your entire database using nothing but the log files, so these files are important for catastrophic database recovery. But typically, you'll want to archive the log files that are no longer in use by Berkeley DB, and then remove them from disk to conserve space. Berkeley DB provides a **db_archive** utility for, among other things, listing the log files that are associated with a given database and which are no longer in use. That way, you know which files to archive and remove.

Subversion's own repository uses a `post-commit` hook script, which, after performing a ˆhot backup˜ of the repository, removes these excess logfiles. In the Subversion source tree, the script `tools/backup/hot-backup.py` illustrates the safe way to perform a backup of a Berkeley DB database environment while it's being actively ac−

cessed: recursively copy the entire repository directory, then re−copythe logfiles listed by **db_archive −l**

Generally speaking, only the truly paranoid would need to backup their entire repository every time a commit oc− curred. Subversion's repository is truly paranoid for what should be obvious reasons! However, assuming that a given repository has some other redundancy mechanism in place with relatively fine granularity (per−commit, for example), a hot backup of the database might be something that a repository administrator would want to include as part of a system−widenightly backup. For more repositories, archived commit emails alone are sufficient restoration sources, at least for the last few commits. But it's your data; protect it as much as you'd like.

Berkeley DB also comes with a pair of utilities for converting the database tables to and from flat ASCII text files. The **db_dump** and **db_load** programs write and read, respectively, a custom file format which describes the keys and values in a Berkeley DB database. Since Berkeley databases are not portable across machine architectures, this format is a useful way to transfer those databases from machine to machine, irrespective of architecture or operating system.

Repository Cleanup

Your Subversion repository will generally require very little attention once it is configured to your liking. However, there are times when some manual assistance from an administrator might be in order. The **svnadmin** utility pro− vides some helpful functionality to assist you in performing such tasks as

- modifying commit log messages,
- removing dead transactions,
- recovering ˆwedged˜ repositories, and
- migrating repository contents to a different repository.

Perhaps the most commonly used of **svnadmin**'s subcommands is `setlog`. When a transaction is committed to the repository and promoted to a revision, the descriptive log message associated with that new revision (and provided by the user) is stored as an unversioned property attached to the revision itself. In other words, the repository re− members only the latest value of the property, and discards previous ones.

Sometimes a user will have an error in her log message (a misspelling or some misinformation, perhaps). If the repository is configured (using the `pre-revprop-change` and `post-revprop-change` hooks; see the section called ˆHook scripts˜) to accept changes to this log message after the commit is finished, then the user can ˆfix˜ her log message remotely using the **svn** program's `propset` command (see Chapter 8). However, because of the poten− tial to lose information forever, Subversion repositories are not, by default, configured to allow changes to unver− sioned properties˙ except by an administrator.

If a log message needs to be changed by an administrator, this can be done using **svnadmin setlog**. This command changes the log message (the `svn:log` property) on a given revision of a repository, reading the new value from a provided file.

```
$ echo "Here is the new, correct log message" > newlog.txt
$ svnadmin setlog myrepos newlog.txt -r 388
```

Don't Lose That Log Message!

Administrators should be aware that using **svnadmin setlog** bypasses any revision property hook scripts that might be active for the repository. Be extremely careful with this subcommand, ensuring that you are providing the correct revision number for the change.

Another common use of **svnadmin** is to query the repository for outstanding˙ possibly dead˙ Subversion transac−

tions. In the event that commit should fail, the transaction is usually cleaned up. That is, the transaction itself is removed from the repository, and any data associated with (and only with) that transaction is removed as well. Occasionally, though, a failure occurs in such a way that the cleanup of the transaction never happens. This could happen for several reasons: perhaps the client operation was inelegantly terminated by the user, or a network failure might have occurred in the middle of an operation, etc. Regardless of the reason, these dead transactions serve only to clutter the repository and consume resources.

You can use **svnadmin**'s `lstxns` command to list the names of the currently outstanding transactions.

```
$ svnadmin lstxns myrepos
19
3a1
a45
$
```

Each item in the resultant output can then be used with **svnlook** (and its `--transaction` option) to determine who created the transaction, when it was created, what types of changes were made in the transaction˙ in other words, whether or not the transaction is a safe candidate for removal! If so, the transaction's name can be passed to **svnadmin rmtxns**, which will perform the cleanup of the transaction. In fact, the `rmtxns` subcommand can take its input directly from the output of `lstxns`!

```
$ svnadmin lstxns myrepos | svnadmin rmtxns myrepos
$
```

If you use these two subcommands like this, you should consider making your repository temporarily inaccessible to clients. That way, no one can begin a legitimate transaction before you start your cleanup. The following is a little bit of shell–scriptingthat can quickly generate information about each outstanding transaction in your repository:

Example 5.1. Reporting outstanding transactions

```
#!/bin/sh

### Generate informational output for all outstanding transactions in
### a Subversion repository

SVNADMIN=/usr/local/bin/svnadmin
SVNLOOK=/usr/local/bin/svnlook

REPOS=${1}
if [ x$REPOS = x ] ; then
  echo "usage: $0 REPOS_PATH"
  exit
fi

for TXN in `${SVNADMIN} lstxns ${REPOS}`; do
  echo "---[ Transaction ${TXN} ]-------------------------------------------"
  ${SVNLOOK} info ${REPOS} --transaction ${TXN}
done
```

You can run the previous script using **/path/to/script /path/to/repos**. The output is basically a concatenation of several chunks of **svnlook info** output (see the section called ˆsvnlook˜), and will look something like:

```
$ txn-info.sh myrepos
---[ Transaction 19 ]-------------------------------------------
sally
2001-09-04 11:57:19 -0500 (Tue, 04 Sep 2001)
0
---[ Transaction 3a1 ]-------------------------------------------
harry
2001-09-10 16:50:30 -0500 (Mon, 10 Sep 2001)
```

```
39
Trying to commit over a faulty network.
---[ Transaction a45 ]-------------------------------------------
sally
2001-09-12 11:09:28 -0500 (Wed, 12 Sep 2001)
0
$
```

Usually, if you see a dead transaction that has no log message attached to it, this is the result of a failed update (or update–like) operation. These operations use Subversion transactions under the hood to mimic working copy state. Since they are never intended to be committed, Subversion doesn't require a log message for those transactions. Transactions that do have log messages attached are almost certainly failed commits of some sort. Also, a transaction's datestamp can provide interesting information˙ for example, how likely is it that an operation begun nine months ago is still active?

In short, transaction cleanup decisions need not be made unwisely. Various sources of information˙ including Apache's error and access logs, the logs of successful Subversion commits, and so on˙ can be employed in the decision–making process. Finally, an administrator can often simply communicate with a seemingly dead transaction's owner (via email, for example) to verify that the transaction is, in fact, in a zombie state.

Repository recovery

In order to protect the data in your repository, the database back–end uses a locking mechanism. This mechanism ensures that portions of the database are not simultaneously modified by multiple database accessors, and that each process sees the data in the correct state when that data is being read from the database. When a process needs to change something in the database, it first checks for the existence of a lock on the target data. If the data is not locked, the process locks the data, makes the change it wants to make, and then unlocks the data. Other processes are forced to wait until that lock is removed before they are permitted to continue accessing that section of the database.

In the course of using your Subversion repository, fatal errors (such as running out of disk space or available memory) or interruptions can prevent a process from having the chance to remove the locks it has placed in the database. The result is that the back–end database system gets ˆwedged˜. When this happens, any attempts to access the repository hang indefinitely (since each new accessor is waiting for a lock to go away˙ which isn't going to happen).

First, if this happens to your repository, don't panic. Subversion's filesystem takes advantage of database transactions and checkpoints and pre–write journaling to ensure that only the most catastrophic of events [3] can permanently destroy a database environment. A sufficiently paranoid repository administrator will be making off–site backups of the repository data in some fashion, but don't call your system administrator to restore a backup tape just yet.

Secondly, use the following recipe to attempt to ˆunwedge˜ your repository:

1. Make sure that there are no processes accessing (or attempting to access) the repository. For networked repositories, this means shutting down the Apache HTTP Server, too.

2. Become the user who owns and manages the repository.

3. Run the command **svnadmin recover /path/to/repos**. You should see output like this:

   ```
   Acquiring exclusive lock on repository db, and running recovery procedures.
   Please stand by...
   Recovery completed.
   The latest repos revision is 19.
   ```

4. Restart the Subversion server.

This procedure fixes almost every case of repository lock–up. Make sure that you run this command as the user that owns and manages the database, not just as `root`. Part of the recovery process might involve recreating from scratch

3 e.g.: hard drive + huge electromagnet = disaster

various database files (shared memory regions, for example). Recovering as `root` will create those files such that they are owned by `root`, which means that even after you restore connectivity to your repository, regular users will be unable to access it.

If the previous procedure, for some reason, does not successfully unwedge your repository, you should do two things. First, move your broken repository out of the way and restore your latest backup of it. Then, send an email to the Subversion developer list (at `<dev@subversion.tigris.org>`) describing your problem in detail. Data integrity is an extremely high priority to the Subversion developers.

Migrating a repository

A Subversion filesystem has its data spread throughout various database tables in a fashion generally understood by (and of interest to) only the Subversion developers themselves. However, circumstances may arise that call for all, or some subset, of that data to be collected into a single, portable, flat file format. Subversion provides such a mechanism, implemented in a pair of **svnadmin** subcommands: `dump` and `load`.

The most common reason to dump and load a Subversion repository is due to changes in Subversion itself. As Subversion matures, there are times when certain changes made to the back-end database schema cause Subversion to be incompatible with previous versions of the repository. The recommended course of action when you are upgrading across one of those compatibility boundaries is a relatively simple process:

1. Using your *current* version of **svnadmin**, dump your repositories to dump files.
2. Upgrade to the new version of Subversion.
3. Move your old repositories out of the way, and create new empty ones in their place using your *new* **svnadmin**.
4. Again using your *new* **svnadmin**, load your dump files into their respective, just-created repositories.
5. Finally, be sure to copy any customizations from your old repositories to the new ones, including `DB_CONFIG` files and hook scripts. You'll want to pay attention to the release notes for the new release of Subversion to see if any changes since your last upgrade affect those hooks or configuration options.

svnadmin dump will output a range of repository revisions that are formatted using Subversion's custom filesystem dump format. The dump format is printed to the standard output stream, while informative messages are printed to the standard error stream. This allows you to redirect the output stream to a file while watching the status output in your terminal window. For example:

```
$ svnadmin youngest myrepos
26
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.

* Dumped revision 25.
* Dumped revision 26.
```

At the end of the process, you will have a single file (`dumpfile` in the previous example) that contains all the data stored in your repository in the requested range of revisions.

The other subcommand in the pair, **svnadmin load**, parses the standard input stream as a Subversion repository dump file, and effectively replays those dumped revisions into the target repository for that operation. It also gives informative feedback, this time using the standard output stream.

```
$ svnadmin load newrepos < dumpfile
<<< Started new txn, based on original revision 1
     * adding path : A ... done.
```

```
     * adding path : A/B ... done.

------- Committed new rev 1 (loaded from original rev 1) >>>

<<< Started new txn, based on original revision 2
     * editing path : A/mu ... done.
     * editing path : A/D/G/rho ... done.

------- Committed new rev 2 (loaded from original rev 2) >>>


<<< Started new txn, based on original revision 25
     * editing path : A/D/gamma ... done.

------- Committed new rev 25 (loaded from original rev 25) >>>

<<< Started new txn, based on original revision 26
     * adding path : A/Z/zeta ... done.
     * editing path : A/mu ... done.

------- Committed new rev 26 (loaded from original rev 26) >>>
```

Note that because **svnadmin** uses standard input and output streams for the repository dump and load process, people who are feeling especially saucy can try things like this (perhaps even using different versions of **svnadmin** on each side of the pipe):

```
$ svnadmin create newrepos
$ svnadmin dump myrepos | svnadmin load newrepos
```

We mentioned previously that **svnadmin dump** outputs a range of revisions. Use the `--revision` option to specify a single revision to dump, or a range of revisions. If you omit this option, all the existing repository revisions will be dumped.

```
$ svnadmin dump myrepos --revision 23 > rev-23.dumpfile
$ svnadmin dump myrepos --revision 100:200 > revs-100-200.dumpfile
```

As Subversion dumps each new revision, it outputs only enough information to allow a future loader to re−createthat revision based on the previous one. In other words, for any given revision in the dump file, only the items that were changed in that revision will appear in the dump. The only exception to this rule is the first revision that is dumped with the current **svnadmin dump** command.

By default, Subversion will not express the first dumped revision as merely differences to be applied to the previous revision. For one thing, there is no previous revision in the dump file! And secondly, Subversion cannot know the state of the repository into which the dump data will be loaded (if it ever, in fact, occurs). To ensure that the output of each execution of **svnadmin dump** is self−sufficient,the first dumped revision is by default a full representation of every directory, file, and property in that revision of the repository.

However, you can change this default behavior. If you add the `--incremental` option when you dump your repository, **svnadmin** will compare the first dumped revision against the previous revision in the repository, the same way it treats every other revision that gets dumped. It will then output the first revision exactly as it does the rest of the revisions in the dump range˙ mentioning only the changes that occurred in that revision. The benefit of this is that you can create several small dump files that can be loaded in succession, instead of one large one, like so:

```
$ svnadmin dump myrepos 0 1000 > dumpfile1
$ svnadmin dump myrepos 1001 2000 --incremental > dumpfile2
$ svnadmin dump myrepos 2001 3000 --incremental > dumpfile3
```

These dump files could be loaded into a new repository with the following command sequence:

```
$ svnadmin load newrepos < dumpfile1
```

```
$ svnadmin load newrepos < dumpfile2
$ svnadmin load newrepos < dumpfile3
```

Another neat trick you can perform with this `--incremental` option involves appending to an existing dump file a new range of dumped revisions. For example, you might have a `post-commit` hook that simply appends the repository dump of the single revision that triggered the hook. Or you might have a script like the following that runs nightly to append dump file data for all the revisions that were added to the repository since the last time the script ran.

Example 5.2. Using incremental repository dumps

```
#!/usr/bin/perl

$repos_path  = '/path/to/repos';
$dumpfile    = '/usr/backup/svn-dumpfile';
$last_dumped = '/var/log/svn-last-dumped';

# Figure out the starting revision (0 if we cannot read the last-dumped file,
# else use the revision in that file incremented by 1).
if (open LASTDUMPED, "$last_dumped")
{
    $new_start = <LASTDUMPED>;
    chomp $new_start;
    $new_start++;
    close LASTDUMPED;
}
else
{
    $new_start = 0;
}

# Query the youngest revision in the repos.
$youngest = `svnlook youngest $repos_path`;
chomp $youngest;

# Do the backup.
`svnadmin dump $repos_path $new_start $youngest --incremental >> $dumpfile`;

# Store a new last-dumped revision
open LASTDUMPED, "> $last_dumped" or die;
print LASTDUMPED "$youngest\n";
close LASTDUMPED;

# All done!
```

Used like this, **svnadmin**'s `dump` and `load` commands can be a valuable means by which to backup changes to your repository over time in case of a system crash or some other catastrophic event.

Finally, another possible use of the Subversion repository dump file format is conversion from a different storage mechanism or version control system altogether. Because the dump file format is, for the most part, human–readable, 4 it should be relatively easy to describe generic sets of changes˙ each of which should be treated as a new revision˙ using this file format.

Networking a Repository

A Subversion repository can be accessed simultaneously by clients running on the same machine on which the repository resides. But the typical Subversion setup involves a single server machine being accessed from clients on computers all over the office˙ or, perhaps, all over the world. This section describes how to get your Subversion repository exposed outside its host machine for use by remote clients.

4The Subversion repository dump file format resembles an RFC–822format, the same type of format used for most email.

Subversion's primary network server is the Apache HTTP Server (**httpd**), speaking the WebDAV/deltaV protocol. This protocol (an extension to HTTP 1.1; see `http://www.webdav.org/`) takes the ubiquitous HTTP protocol that is core of the World Wide Web, and adds writing˙ specifically, versioned writing˙ capabilities. The result is a standardized, robust system that is conveniently packaged as part of the Apache 2.0 software, is supported by numerous pieces of core operating system and third–party products, and which doesn't require network administrators to open up yet another custom port. 5

Much the following discussion includes references to Apache configuration directives. While some examples are given of the use of these directives, describing them in full is outside the scope of this chapter. The Apache team maintains excellent documentation, publicly available on their website at `http://httpd.apache.org`. For example, a general reference for the configuration directives is located at `http://httpd.apache.org/docs-2.0/mod/directives.html`.)

Also, as you make changes to your Apache setup, it is likely that somewhere along the way a mistake will be made. If you are not already familiar with Apache's logging subsystem, you should become aware of it. In your `httpd.conf` file are directives which specify the on–disk locations of the access and error logs generated by Apache (the `CustomLog` and `ErrorLog` directives, respectively). Subversion's mod_dav_svn uses Apache's error logging interface as well. You can always browse the contents of those files for information that might reveal the source of a problem which is not clearly noticeable otherwise.

What You Need to Network Your Repository

To network your repository, you basically need four components, available in two packages. You'll need Apache **httpd** 2.0, the **mod_dav** DAV module that comes with it, Subversion, and the **mod_dav_svn** filesystem provider module distributed with Subversion. Once you have all of those components, the process of networking your repository is as simple as:

- getting httpd 2.0 up and running with the mod_dav module,
- installing the mod_dav_svn plugin to mod_dav, which uses Subversion's libraries to access the repository, and
- configuring your `httpd.conf` file to export (or expose) the repository.

You can accomplish the first two items either by compiling **httpd** and Subversion from source code, or by installing pre–built binary packages of them on your system. For the most up–to–date information on how to compile Subversion for use with Apache HTTP Server, as well as how to compile and configure Apache itself for this purpose, see the `INSTALL` file in the top level of the Subversion source code tree.

Basic Apache Configuration

Once you have all the necessary components installed on your system, all that remains is the configuration of Apache via its `httpd.conf` file. Instruct Apache to load the mod_dav_svn module using the `LoadModule` directive. This directive must precede any other Subversion–related configuration items. If your Apache was installed using the default layout, your **mod_dav_svn** module should have been installed in the `modules` subdirectory of the Apache install location (often `/usr/local/apache2`). The `LoadModule` directive has a simple syntax, mapping a named module to the location of a shared library on disk:

```
LoadModule dav_svn_module     modules/mod_dav_svn.so
```

Note that if **mod_dav** was compiled as a shared object (instead of statically linked, directly to the **httpd** binary), you'll need a similar `LoadModule` statement for it, too.

At a later location in your configuration file, you now need to tell Apache where you keep your Subversion repository (or repositories). The `Location` directive has an XML–like notation, starting with an opening tag, and ending with a closing tag, with various other configuration directives in the middle. The purpose of the `Location` directive

5 They really hate doing that.

is to instruct Apache to do something special when handling requests that are directed at a given URL or one of its children. In the case of Subversion, you want Apache to simply hand off support for URLs that point at versioned resources to the DAV layer. You can instruct Apache to delegate the handling of all URLs whose path portions (the part of the URL that follows the server's name and the optional port number) begin with `/repos/` to a DAV provider whose repository is located at `/absolute/path/to/repository` using the following `httpd.conf` syn–tax:

```
<Location /repos>
  DAV svn
  SVNPath /absolute/path/to/repository
</Location>
```

If you plan to support multiple Subversion repositories that will reside in the same parent directory on your local disk, you can use an alternative directive, the `SVNParentPath` directive, to indicate that common parent directory. For example, if you know you will be creating multiple Subversion repositories in a directory `/usr/local/svn` that would be accessed via URLs like `http://my.server.com/svn/repos1`, `http://my.server.com/svn/repos2`, and so on, you could use the following `httpd.conf` configuration syn–tax:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
</Location>
```

Using the previous syntax, Apache will delegate the handling of all URLs whose path portions begin with `/svn/` to the Subversion DAV provider, which will then assume that any items in the directory specified by the `SVNParent–Path` directive are actually Subversion repositories. This is a particularly convenient syntax in that, unlike the use of the `SVNPath` directive, you don't have to restart Apache in order to create and network new repositories.

Permissions, Authentication, and Authorization

Now, at this stage, you should strongly consider the question of permissions. If you've been running Apache for some time now as your regular web server, you probably already have a collection of content˙ web pages, scripts and such. These items have already been configured with a set of permissions that allows them to work with Apache, or more appropriately, that allows Apache to work with those files. Apache, when used a Subversion server, will also need the correct permissions to read and write to your Subversion repository.

You will need to determine a permission system setup that satisfies Subversion's requirements without messing up any previously existing web page or script installations. This might mean changing the permissions on your Subver–sion repository to match those in use by other things the Apache serves for you, or it could mean using the `User` and `Group` directives in `httpd.conf` to specify that Apache should run as the user and group that owns your Subver–sion repository. There is no single correct way to set up your permissions, and each administrator will have different reasons for doing things a certain way. Just be aware that permission–relatedproblems are perhaps the most common oversight when configuring a Subversion repository for use with Apache.

And while we are speaking about permissions, we should address how the authorization and authentication mecha–nisms provided by Apache fit into the scheme of things. Unless you have some system–wideconfiguration of these things, the Subversion repositories you make available via the `Location` directives will be generally accessible to everyone. In other words,

- anyone can use their Subversion client to checkout a working copy of a repository URL (or any of its subdirecto–ries),
- anyone can interactively browse the repository's latest revision simply by pointing their web browser to the repository URL, and
- anyone can commit to the repository.

If you want to restrict either read or write access to a repository as a whole, you can use Apache's built–in access control features. The easiest such feature is the Basic authentication mechanism, which simply uses a username and password to verify that a user is who she says she is. Apache provides an **htpasswd** utility for managing the list of acceptable usernames and passwords, those to whom you wish to grant special access to your Subversion repository. Let's grant commit access to Sally and Harry. First, we need to add them to the password file.

```
$ ### First time: use -c to create the file
$ htpasswd -c /etc/svn-auth-file harry
New password: *****
Re-type new password: *****
Adding password for user harry
$ htpasswd /etc/svn-auth-file sally
New password: *******
Re-type new password: *******
Adding password for user sally
$
```

Next, you need to add some more `httpd.conf` directives inside your `Location` block to tell Apache what to do with your new password file. The `AuthType` directive specifies the type of authentication system to use. In this case, we want to specify the `Basic` authentication system. `AuthName` is an arbitrary name that you give for the au–thentication domain. Most browsers will display this name in the pop–up dialog box when the browser is querying the user for his name and password. Finally, use the `AuthUserFile` directive to specify the location of the pass–word file you created using **htpasswd**.

After adding these three directives, your `<Location>` block should look something like this:

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
</Location>
```

Now, at this stage, if you were to restart Apache, any Subversion operations which required authentication would harvest a username and password from the Subversion client, which would either provide previously cached values for these things, or prompt the user for the information. All that remains is to tell Apache which operations actually require that authentication.

You can restrict access on all repository operations by adding the `Require valid-user` directive to your `<Location>` block. Using our previous example, this would mean that only clients that claimed to be either `harry` or `sally`, and which provided the correct password for their respective username, would be allowed to do anything with the Subversion repository.

Sometimes you don't need to run such a tight ship. The repository at `http://svn.collab.net/repos/svn` which holds the Subversion source code, for example, allows anyone in the world to perform read–only repository tasks (like checking out working copies and browsing the repository with a web browser), but restricts all write operations to authenticated users. To do this type of selective restriction, you can use the `Limit` and `LimitExcept` configuration directives. Like the `Location` directive, these blocks have starting and ending tags, and you would nest them inside your `<Location>` block.

The parameters present on the `Limit` and `LimitExcept` directives are HTTP request types that are affected by that block. For example, if you wanted to disallow all access to your repository except the currently supported read–only operations, you would use the `LimitExcept` directive, passing the `GET`, `PROPFIND`, `OPTIONS`, and `REPORT` request type parameters. Then the previously mentioned `Require valid-user` directive would be placed inside the `<LimitExcept>` block instead of just inside the `<Location>` block.

```
<Location /svn>
  DAV svn
  SVNParentPath /usr/local/svn
```

```
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile /path/to/users/file
  <LimitExcept GET PROPFIND OPTIONS REPORT>
    Require valid-user
  </LimitExcept>
</Location>
```

These are only a few simple examples. For more in-depth information about Apache access control, take a look at the `Security` section of the Apache documentation's tutorials collection at `http://httpd.apache.org/docs-2.0/misc/tutorials.html`.

Server Names and the COPY Request

Subversion makes use of the `COPY` request type to perform server-side copies of files and directories. As part of the sanity checking done by the Apache modules, the source of the copy is expected to be located on the same machine as the destination of the copy. To satisfy this requirement, you might need to tell mod_dav the name you use as the hostname of your server. Generally, you can use the `ServerName` directive in `httpd.conf` to accomplish this.

```
ServerName svn.red-bean.com
```

If you are using Apache's virtual hosting support via the `NameVirtualHost` directive, you may need to use the `ServerAlias` directive to specify additional names that your server is known by. Again, refer to the Apache documentation for full details.

Miscellaneous Apache Features

Several of the features already provided by Apache in its role as a robust Web server can be leveraged for increased functionality or security in Subversion as well. Subversion communicates with Apache using Neon, which is a generic HTTP/WebDAV library with support for such mechanisms as SSL (the secure socket layer) and Deflate compression (the same algorithm used by the **gzip** and **PKZIP** programs to ˆshrink˜ files into smaller chunks of data). You need only to compile support for the features you desire into Subversion and Apache, and properly configure the programs to use those features.

This means that SSL-enabled Subversion clients can access SSL-enabled Apache servers and perform all communication using an encrypted protocol, all by using `https:` URLs with their Subversion clients instead of `http:` ones. Businesses that need to expose their repositories for access outside the company firewall should be conscious of the possibility that unauthorized parties could be ˆsniffing˜ their network traffic. SSL makes that kind of unwanted attention less likely to result in sensitive data leaks. Apache can be configured such that only SSL-enabled Subversion clients can communicate with the repository.

Deflate compression places a small burden on the client and server to compress and decompress network transmissions as a way to minimize the size of the actual transmission. In cases where network bandwidth is in short supply, this kind of compression can greatly increase the speed at which communications between server and client can be sent. In extreme cases, this minimized network transmission could be the difference between an operation timing out or completing successfully.

Less interesting, but equally useful, are other features of the Apache and Subversion relationship, such as the ability to specify a custom port (instead of the default HTTP port 80) or a virtual domain name by which the Subversion repository should be accessed, or the ability to access the repository through a proxy. These things are all supported by Neon, so Subversion gets that support for free.

Adding projects

Once your repository is created and configured, all that remains is to begin using it. If you have a collection of existing data that is ready to be placed under version control, you will more than likely want to use the **svn** client program's `import` subcommand to accomplish that. Before doing this, though, you should carefully consider your long-term plans for the repository. In this section, we will offer some advice on how to plan the layout of your repository, and how to get your data arranged in that layout.

Choosing a Repository Layout

While Subversion allows you to move around versioned files and directories without any loss of information, doing so can still disrupt the workflow of those who access the repository often and come to expect things to be at certain locations. Try to peer into the future a bit; plan ahead before placing your data under version control. By ^laying out˜ the contents of your repositories in an effective manner the first time, you can prevent a load of future headaches.

There are a few things to consider when setting up Subversion repositories. Let's assume that as repository administrator, you will be responsible for supporting the version control system for several projects. The first decision is whether to use a single repository for multiple projects, or to give each project its own repository, or some compromise of these two.

There are benefits to using a single repository for multiple projects, most obviously the lack of duplicated maintenance. A single repository means that there is one set of hook scripts, one thing to routinely backup, one thing to dump and load if Subversion releases an incompatible new version, and so on. Also, you can easily move data between projects easily, and without losing any historical versioning information.

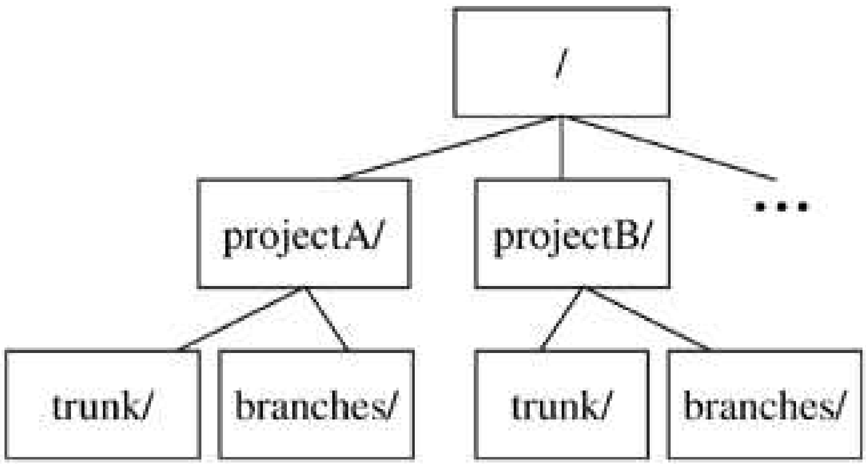
The downside of using a single repository is that different projects may have different commit mailing lists or different authentication and authorization requirements. Also, remember that Subversion uses repository−global revision numbers. Some folks don't like the fact that even though no changes have been made to their project lately, the youngest revision number for the repository keeps climbing because other projects are actively adding new revisions.

A middle−ground approach can be taken, too. For example, projects can be grouped by how well they relate to each other. You might have a few repositories with a handful of projects in each repository. That way, projects that are likely to want to share data can do so easily, and as new revisions are added to the repository, at least the developers know that those new revisions are at least remotely related to everyone who uses that repository.

After deciding how to organize your projects with respect to repositories, you'll probably want to think about directory hierarchies in the repositories themselves. Because Subversion uses regular directory copies for branching and tagging (see Chapter 4), the Subversion community recommends using one of two different approaches here. Both approaches employ the use of directories named `trunk`, meaning the directory under which the main project development occurs, and `branches`, which is a directory in which to create various named branches of the main development line.

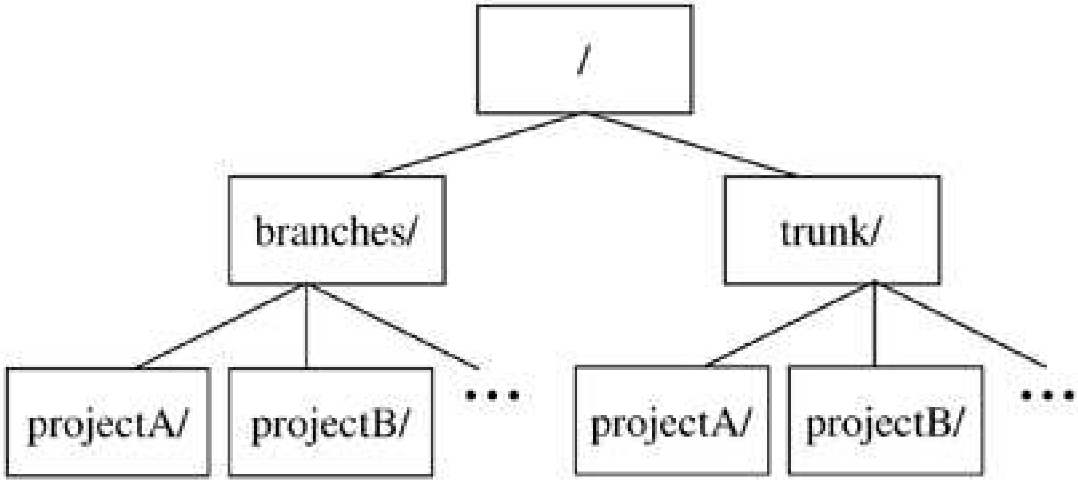
The first is to place each project in a subdirectory of the root filesystem directory, with `trunk` and `branches` directories immediately under each project directory, as demonstrated in Figure 5−1.

Figure 5.1. One suggested repository layout.



The second is do the reverse˙ to have the `trunk` and `branches` directories immediately in the top level of the filesystem, each with subdirectories for all the projects in the repository, as demonstrated in Figure 5−2.

Figure 5.2. Another suggested repository layout.



Lay out your repository in whatever way you see fit. Subversion does not expect or enforce a layout schema˙ in its eyes, a directory is a directory is a directory. Ultimately, you should choose the repository arrangement that meets your needs and those of the projects that live there.

Creating the Layout, and Importing Initial Data

After deciding how to arrange the projects in your repository, you'll probably want to actually populate the repository with that layout and with initial project data. There are a couple of ways to do this in Subversion. You could use the **svn mkdir** command (see the section called ˆsvn mkdir˜) to create each directory in your skeletal repository layout, one−by−one.A quicker way to accomplish the same task is to use the **svn import** command (see the section called ˆsvn import˜. By first creating the layout in a temporary location on your drive, you can import the whole layout tree into the repository in a single commit:

```
$ mkdir tmpdir
$ cd tmpdir
$ mkdir projectA
$ mkdir projectA/trunk
$ mkdir projectA/branches
$ mkdir projectB
$ mkdir projectB/trunk
$ mkdir projectB/branches

$ svn import file:///path/to/repos . −−message 'Initial repository layout'
Adding         projectA
Adding         projectA/trunk
Adding         projectA/branches
Adding         projectB
Adding         projectB/trunk
Adding         projectB/branches

Committed revision 1.
$ cd ..
$ rm −rf tmpdir
$
```

Once you have your skeletal layout in place, you can begin importing actual project data into your repository, if any such data exists yet. Once again, there are several ways to do this. You could use the **svn import** command. You could checkout a working copy from your new repository, move and arrange project data inside the working copy, and use the **svn add** and **svn commit** commands. But once we start talking about such things, we're no longer discussing repository administration. If you aren't already familiar with the **svn** client program, see Chapter 3.

Chapter 6. Advanced Topics

Run−timeConfiguration Area

When you first run the **svn** command−lineclient, it creates a per−userconfiguration area. On Unix−likesystems, a `.subversion/` directory is created in the user's home directory. On Win32 systems, a `Subversion` folder is cre− ated wherever its appropriate to do so (typically somewhere within `Documents and Settings\username`, al− though it depends on the system.)

Proxies

At the time of writing, the configuration area only contains one item: a `proxies` file. By setting values in this file, your Subversion client can operate through an http proxy. (Read the file itself for details; it should be self− documenting.)

Config

Soon˙ very soon˙ a `config` file will exist in this area for defining general user preferences. For example, the pre− ferred **$EDITOR** to use, options to pass through to **svn diff**, preferences for date/time formats, and so on.

Multiple config areas

On Unix, an administrator can create ˆglobal˜ Subversion preferences by creating and populating an `/etc/subversion/` area. The per−user`~/.subversion/` configuration will still override these defaults, however.

On Win32, an administrator has the option of creating three other locations: a global `Subversion` folder in the ˆAll Users˜ area, a collection of global registry settings, or a collection of per−userregistry settings. The registry settings are set in:

```
HKCU\Software\Tigris.org\Subversion\Proxies
HKCU\Software\Tigris.org\Subversion\Config
etc.
```

To clarify, here is the order Subversion searches for run−timesettings on Win32. Each subsequent location overrides the previous one:

- global registry
- global `Subversion` folder
- user registry
- user `Subversion` folder

Properties

Subversion allows you to attach arbitrary ˆmetadata˜ to files and directories. We refer to this data as properties, and they can be thought of as collections of name/value pairs (hash−tables)attached to each item in your working copy.

To set or get a property on a file or directory, use the **svn propset** and **svn propget** commands. To list all properties attached to an item, use **svn proplist**. To delete a property, use **svn propdel**.

```
$ svn propset color green foo.c
```

```
property `color' set on 'foo.c'

$ svn propget color foo.c
green

$ svn propset height "5 feet" foo.c
property `height' set on 'foo.c'

 svn proplist foo.c
Properties on 'foo.c':
  height
  color

$ svn proplist foo.c --verbose
Properties on 'foo.c':
  height : 5 feet
  color : green

$ svn propdel color foo.c
property `color' deleted from 'foo.c'
```

Properties are *versioned*, just like file contents. This means that new properties can be merged into your working files, and can sometimes come into conflict too. Property values need not be text, either. For example, you could attach a binary property–valueby using the `-F` switch:

```
$ svn propset x-face -F joeface.jpg foo.c
property `x-face' set on 'foo.c'
```

Subversion also provides a great convenience method for editing existing properties: **svn propedit**. When you invoke it, Subversion will open the value of the property in question in your favorite editor (or at least the editor that you've defined as **$EDITOR** in your shell), and you can edit the value just as you would edit any text file. This is exceptionally convenient for properties that are a newline–separatedarray of values. (More about this later).

Property changes are still considered ˆlocal modifications˜, and aren't permanent until you commit. Like textual changes, property changes can be seen by **svn diff**, **svn status**, and reverted altogether with **svn revert**:

```
$ svn diff
Property changes on: foo.c
___________________________________________________________________
Name: color
   + green

$ svn status
_M   foo.c
```

Notice that a 2nd column has appeared in the status output; the leading underscore indicates that you've not made any textual changes, but the `M` means you've modified the properties. **svn status** tries to hide the 2nd ˆproperty˜ olcumn when an item has no properties at all; this was a design choice, to ease new users into the concept. When properties are created, edited, or updated on an item, that 2nd column appears forever after.

Also: don't worry about the non–standardway that Subversion currently displays property differences. You can still run **svn diff** and redirect the output to create a usable patch file. The **patch** program will ignore property patches; as a rule, it ignores any noise it can't understand. (In future versions of Subversion, though, we may start using a new patch format that describes property changes and file copies/renames.)

Special properties

Subversion has no particular policy regarding properties; they can be used for any purpose. The only restriction is that Subversion has reserved the `svn:` name prefix for itself. A number of special ˆmagic˜ properties begin with this prefix. We'll cover these features here.

svn:executable

This is a file–only property, and can be set to any value. Its mere existence causes a file's permissions to be executable.

svn:mime-type

At the present time, Subversion examines the `svn:mime-type` property to decide if a file is text or binary. If the file has no `svn:mime-type` property, or if the property's value matches `text/*`, then Subversion assumes it is a text file. If the file has the `svn:mime-type` property set to anything other than `text/*`, it assumes the file is binary.

If Subversion believes that the file is binary, it will not attempt to perform contextual merges during updates. Instead, Subversion creates two files side–by–side in your working copy; the one containing your local modifications is renamed with a `.orig` extension.

Subversion also helps users by running a binary–detection algorithm in the **svn import** and **svn add** subcommands. These subcommands try to make a good guess at a file's binary–ness, and then (possibly) set a `svn:mime-type` property of `application/octet-stream` on the file being added. (If Subversion guesses wrong, you can always remove or hand–edit the property.)

Finally, if the `svn:mime-type` property is set, then mod_dav_svn will use it to fill in the `Content-type:` header when responding to an http GET request. This makes files display more nicely when perusing a repository with a web browser.

svn:ignore

If you attach this property to a directory, it causes certain file patterns within the directory to be ignored by **svn status**. For example, suppose I don't want to see object files or backup files in my status listing:

```
$ svn status
M  ./foo.c
?  ./foo.o
?  ./foo.c~
```

Using **svn propedit**, I would set the value of `svn:ignore` to a newline–delimited list of patterns:

```
$ svn propget svn:ignore .
*.o
*~
```

svn:keywords

Subversion has the ability to substitute useful strings into special keywords within text files. For example, if I placed this text into a file:

```
Here is the latest report from the front lines.
$LastChangedDate$
Cumulus clouds are appearing more frequently as summer approaches.
```

Subversion is able substitute the `$LastChangedDate$` string with the actual date in which this file last changed. The keyword string is not removed in the replacement, just the specific information is placed after the keyword string:

```
Here is the latest report from the front lines.
$LastChangedDate: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) $
```

```
Cumulus clouds are appearing more frequently as summer approaches.
```

Subversion substitutes five keywords

LastChangedDate
: The last time this file changed. Can also be abbreviated as `Date`. The keyword substitution of `$LastChangedDate$` will look something like `$LastChangedDate: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) $`.

LastChangedRevision
: The last revision in which this file changed. Can be abbreviated as `Rev`. The keyword substitution of `$LastChangedRevision` will look something like `$LastChangedRevision: 144 $`.

LastChangedBy
: The last user to change this file. Can be abbreviated as `Author`. The keyword substitution of `$LastChangedBy$` will look something like `$LastChangedBy: joe $`.

HeadURL
: A full URL to the latest version of the file in the repository. Can be abbreviated as `URL`. The keyword substitution of `$HeadURL$` will look something like `$HeadURL: http://svn.collab.net/repos/trunk/README $`.

Id
: A compressed summary of the other keywords, for example: `$Id: bar 148 2002-07-28 21:30:43 epg $`. This means the file `bar` was last changed in revision 148 by committer `epg`, at 2002-07-28 21:30:43.

To activate a keyword, or set of keywords, you merely need to set the `svn:keywords` property to a list of keywords you want replaced. Keywords not listed in `svn:keywords` will not be replaced.

```
$ svn propset svn:keywords "Date Author" foo.c
property 'svn:keywords' set on 'foo.c'
```

And when you commit this property change, you'll discover that all occurrences of `$Date$`, `$LastChangedDate$`, `$Author$`, and `$LastChangedBy$` will have substituted values within `foo.c`.

svn:eol-style

By default, Subversion doesn't pay any attention to line endings. If a text file has either LF, CR, or CRLF endings, then those are the line endings that will exist on the file in both the repository and working copy.

But if developers are working on different platforms, line endings can sometimes become troublesome. For example, if a Win32 developer and Unix developer took turns modifying a file, its line endings might flip-flop back and forth from revision to revision in the repository. This makes examining or merging differences very difficult, as *every* line appears to be changed in each version of the file.

The solution here is to set the `svn:eol-style` property to ''native''. This makes the file always appear with the "native" line endings of each developer's operating system. Note, however, that the file will always contain LF endings in the repository. This prevents the line-ending "churn" from revision to revision.

Alternately, you can force files to always retain a fixed, specific line ending: set a file's `svn:eol-style` property to one of `LF`, `CR` or `CRLF`. A Win32 `.dsp` file, for example, which is used by Microsoft development tools, should always have CRLF endings.

svn:externals

See the section called "Modules".

Modules

Sometimes it's useful to construct a working copy that is made out of a number of different checkouts. For example, you may want different sub–directoriesto come from different locations in a repository.

On the one hand, you could begin by checking out a working copy, and then run **svn switch** on various subdirectories. But this is a bit of work. Wouldn't it be nice to define˙ in a single place˙ exactly how you want the final working copy to be?

This is known as a module. You can define a module by attaching another special ˆmagic˜ `svn:` property to a directory: the `svn:externals` property.

The value of this property is a list of subdirectories and their corresponding URLs:

```
$ svn propget svn:externals projectdir
subdir1/foo       http://url.for.external.source/foo
subdir1/bar       http://blah.blah.blah/repositories/theirproj
subdir1/bar/baz   http://blorg.blorg.blorg/basement/code
```

Assuming that this property is attached to the directory `projectdir`, then when we check it out, we'll get everything else defined by the property.

```
$ svn checkout http://foo.com/repos/projectdir
A  projectdir/blah.c
A  projectdir/gloo.c
A  projectdir/trout.h
Checked out revision 128.

Fetching external item into projectdir/subdir1/foo
A  projectdir/subdir1/foo/rho.txt
A  projectdir/subdir1/foo/pi.txt
A  projectdir/subdir1/foo/tau.doc
Checked out revision 128.
```

By tweaking the value of the `svn:externals` property, the definition of the module can change over time, and subsequent calls to **svn update** will update working copies appropriately.

Vendor branches

Sometimes you want to manage modified third–party source code inside your Subversion repository, while still tracking upstream releases. In CVS this would have been called a ˆvendor branch˜. Subversion doesn't have a formal ˆvendor branch˜, but it is sufficiently flexible that you can still do much the same thing.

The general procedure goes like this. You create a top level directory (we'll use `/vendor`) to hold the vendor branches. Then you import the third party code into a subdirectory of `/vendor`, and copy it into `/trunk` where you make your local changes. With each new release of the code you are tracking you bring it into the vendor branch and merge the changes into `/trunk`, resolving whatever conflicts occur between your local changes and the upstream changes.

Let's try and make this a bit clearer with an example.

First, the initial import.

```
$ svn mkdir http://svnhost/repos/vendor/foobar
$ svn import http://svnhost/repos/vendor/foobar ~/foobar-1.0 current
```

Now we've got the current version of the foobar project in `/vendor/foobar/current`. We make another copy of it so we can always refer to that version, and then copy it into the trunk so you can work on it.

```
$ svn copy http://svnhost/repos/vendor/foobar/current    \
           http://svnhost/repos/vendor/foobar/foobar-1.0 \
           -m 'tagging foobar-1.0'
$ svn copy http://svnhost/repos/vendor/foobar/foobar-1.0 \
           http://svnhost/repos/trunk/foobar             \
           -m 'bringing foobar-1.0 into trunk'
```

Now you just check out a copy of `/trunk/foobar` and get to work!

Later on, the developers at FooBar Widgets, Inc release a new version of their code, so you want to update the version of the code you're using. First, you check out the `/vendor/foobar/current` directory, then copy the new release over that working copy, handle any renames, additions or removals manually, and then commit.

```
$ svn checkout http://svnhost/repos/vendor/foobar/current ~/current
$ cd ~/foobar-1.1
$ tar -cf - . | (cd ~/current ; tar -xf -)
$ cd ~/current
$ mv foobar.c main.c
$ svn move main.c foobar.c
$ svn delete dead.c
$ svn add doc
$ svn add doc/*
$ svn commit -m 'importing foobar 1.1 on vendor branch'
```

Whoa, that was complicated. Don't worry, most cases are far simpler.

What happened? foobar 1.0 had a file called `main.c`. This file was renamed to `foobar.c` in 1.1. So your working-copy had the old `main.c` which Subversion knew about, and the new `foobar.c` which Subversion did not know about. You rename `foobar.c` to `main.c` and **svn mv** it back to the new name. This way, Subversion will know that `foobar.c` is a descendant of `main.c`. `dead.c` has been removed in 1.1, and they have finally written some documentation, so you add that.

Next you copy `/vendor/foobar/current` to `/vendor/foobar/foobar-1.1` so you can always refer back to version 1.1, like this.

```
$ svn copy http://svnhost/repos/vendor/foobar/current    \
           http://svnhost/repos/vendor/foobar/foobar-1.1 \
           -m 'tagging foobar-1.1'
```

Now that you have a pristine copy of foobar 1.1 in `/vendor`, you just have to merge their changes into `/trunk` and you're done. That looks like this.

```
$ svn checkout http://svnhost/repos/trunk/foobar ~/foobar
$ cd ~/foobar
$ svn merge http://svnhost/repos/vendor/foobar/foobar-1.0 \
            http://svnhost/repos/vendor/foobar/foobar-1.1
$
  # resolve all the conflicts between their changes and your changes
$ svn commit -m 'merging foobar 1.1 into trunk'
```

There, you're done. You now have a copy of foobar 1.1 with all your local changes merged into it in your tree.

Vendor branches that have more than several deletes, additions and moves can use the **svn_load_dirs.pl** script that comes with the Subversion distribution. This script automates the above importing steps to make sure that mistakes are minimized. You still need to use the merge commands to merge the new versions of foobar into your own local copy containing your local modifications.

This script has the following enhancements over **svn import**:

- Can be run at any point in time to bring an existing directory in the repository to exactly match an external directory. This script runs all the **svn add**, **svn rm** and optionally any **svn mv** commands as necessary.
- Optionally tag the newly imported directory.
- Optionally add arbitrary properties to files and directories that match a regular expression.

This script takes care of complications where Subversion requires a commit before renaming a file or directory twice, such as if you had a vendor branch that renamed `foobar-1.1/docs/doc.ps` to `foobar-1.2/documents/doc-1.2.ps`. Here, you would rename `docs` to `documents`, perform a commit, then rename `doc.ps` to `doc-1.2.ps`. You could not do the two renames without the commit, because `doc.ps` was already moved once from `docs/doc.ps` to `documents/doc.ps`.

This script always compares the directory being imported to what currently exists in the Subversion repository and takes the necessary steps to add, delete and rename files and directories to make the subversion repository match the imported directory. As such, it can be used on an empty subversion directory for the first import or for any following imports to upgrade a vendor branch.

For the first foobar−1.0release located in `~/foobar-1.0`:

```
$ svn_load_dirs.pl -t foobar-1.0                      \
                   http://svnhost/repos/vendor/foobar \
                   current                            \
                   ~/foobar-1.0
```

svn_load_dirs.pl takes three mandatory arguments. The first argument, `http://svnhost/repos/vendor/foobar`, is the URL to the base Subversion directory to work in. In this case, we're working in the `vendor/foobar` part of the Subversion repository. The next argument, `current`, is relative to the first and is the directory where the current import will take place, in this case `http://svnhost/repos/vendor/foobar/current`. The last argument, `~/foobar-1.0`, is the directory to import. Finally, the optional `-t` command line option is also relative to `http://svnhost/repos/vendor/foobar` and tells **svn_load_dirs.pl** to create a tag of the imported directory in `http://svnhost/repos/vendor/foobar/foobar-1.0`.

The import of foobar−1.1would be taken care of in the same way:

```
$ svn_load_dirs.pl -t foobar-1.1                      \
                   http://svnhost/repos/vendor/foobar \
                   current                            \
                   ~/foobar-1.1
```

The script looks in your current `http://svnhost/repos/vendor/foobar/current` directory and sees what changes need to take place for it to match `~/foobar-1.1`. The script is kind enough to notice that there are files and directories that exist in 1.0 and not in 1.1 and asks if you want to perform any renames. At this point, you can indicate that `main.c` was renamed to `foobar.c` and then indicate that no further renames have taken place.

The script will then delete `dead.c` and add `doc` and `doc/*` to the Subversion repository and finally create a tag foobar−1.1in `http://svnhost/repos/vendor/foobar/foobar-1.1`.

The script also accepts a separate configuration file for applying properties to specific files and directories matching a regular expression that are *added* to the repository. This script will not modify properties of already existing files or directories in the repository. This configuration file is specified to **svn_load_dirs.pl** using the `-p` command line option. The format of the file is either two or four columns.

```
regular_expression control property_name property_value
```

The `regular_expression` is a Perl style regular expression. The `control` column must either be set to `break` or **cont**. It is used to tell **svn_load_dirs.pl** if the following lines in the configuration file should be examined for a match or if all matching should stop. If `control` is set to **break**, then no more lines from the configuration file will be matched. If `control` is set to **cont**, which is short for continue, then more comparisons will be made. Multiple properties can be set for one file or directory this way. The last two columns, `property_name` and `property_value` are optional and are applied to matching files and directories.

If you have whitespace in any of the `regular_expression`, `property_name` or `property_value` columns, you must surround the value with either a single or double quote. You can protect single or double quotes with a \ character. The \ character is removed by this script *only* for whitespace or quote characters, so you do not need to protect any other characters, beyond what you would normally protect for the regular expression.

This sample configuration file was used to load on a Unix box a number of Zip files containing Windows files with `CRLF` end of lines.

```
\.doc$              break   svn:mime-type   application/msword
\.ds(p|w)$          break   svn:eol-style   CRLF
\.ilk$              break   svn:eol-style   CRLF
\.ncb$              break   svn:eol-style   CRLF
\.opt$              break   svn:eol-style   CRLF
\.exe$              break   svn:mime-type   application/octet-stream
dos2unix-eol\.sh$   break
.*                  break   svn:eol-style   native
```

In this example, all the files should be converted to the native end of line style, which the last line of the configuration handles. The exception is **dos2unix-eol.sh**, which contains embedded `CR`'s used to find and replace Windows `CRLF` end of line characters with Unix's `LF` characters. Since **svn** and **svn_load_dirs.pl** convert all `CR`, `CRLF` and `LF` `svn:eol-style` is set to `native`, this file should be left untouched. Hence, the **break** with no property settings.

The Windows Visual C++ and Visual Studio files (`*.dsp`, `*.dsw`, etc.) should retain their `CRLF` line endings on any operating system and any `*.doc` files are always treated as binary files, hence the `svn:mime-type` setting of `application/msword`.

Chapter 7. Developer Information

Subversion is an open-source software project developed under an Apache-style software license. The project is financially backed by CollabNet, Inc., a California-based software development company. The community that has formed around the development of Subversion always welcomes new members who can donate their time and attention to the project. Volunteers are encouraged to assist in any way they can, whether that means finding and diagnosing bugs, refining existing source code, or fleshing out whole new features.

This chapter is for those who wish to assist in the continued evolution of Subversion by actually getting their hands dirty with the source code. We will cover some of the software's more intimate details, the kind of technical nitty-gritty that those developing Subversion itself˙ or writing entirely new tools based on the Subversion libraries˙ should be aware of. If you don't foresee yourself participating with the software at such a level, feel free to skip this chapter with confidence that your experience as a Subversion user will not be affected.

Layered Library Design

Subversion has a modular design, implemented as a collection of C libraries. Each library has a well-defined purpose and interface, and most modules are said to exist in one of three main layers˙ the Repository Layer, the Repository Access (RA) Layer, or the Client Layer. We will examine these layers shortly, but first, see our brief inventory of Subversion's libraries in Table 7-1.For the sake of consistency, we will refer to the libraries by their extensionless Unix library names (e.g.: libsvn_fs, libsvn_wc, mod_dav_svn).

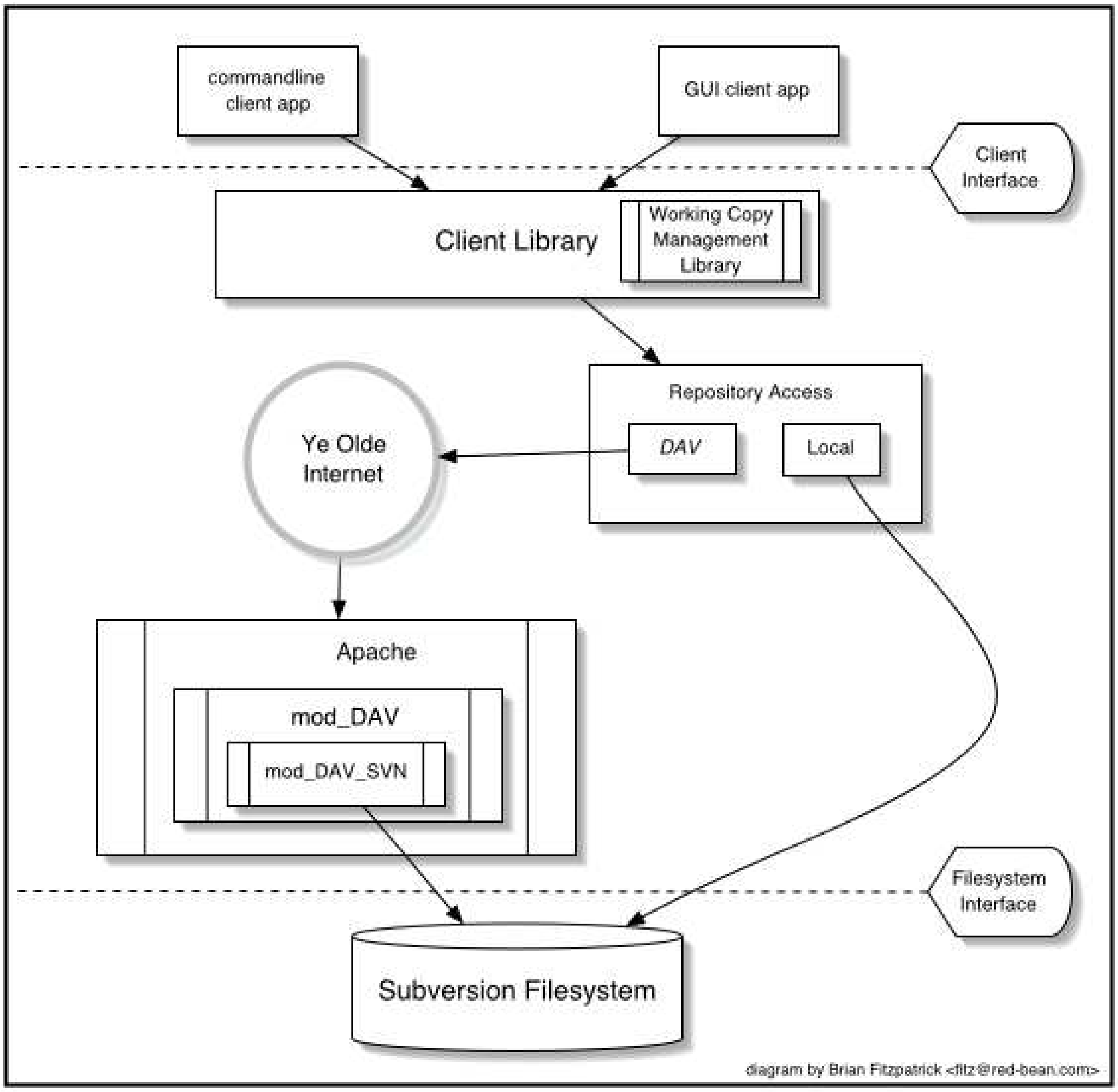
Table 7.1. A brief inventory of the Subversion libraries

Library	Description
libsvn_client	Primary interface for client programs
libsvn_delta	Tree and text differencing routines
libsvn_fs	The Subversion filesystem library
libsvn_ra	Repository Access commons and module loader
libsvn_ra_dav	The WebDAV Repository Access module
libsvn_ra_local	The local Repository Access module
libsvn_repos	Repository interface
libsvn_subr	Miscellaneous helpful subroutines
libsvn_wc	The working copy management library
mod_dav_svn	Apache module for mapping WebDAV operations to Subversion ones

The fact that the word "miscellaneous" only appears once in Table 7-1is a good sign. The Subversion development team is serious about making sure that functionality lives in the right layer and libraries. Perhaps the greatest advantage of the modular design is its lack of complexity from a developer's point of view. As a developer, you can quickly formulate that kind of "big picture" that allows you to pinpoint the location of certain pieces of functionality with relative ease.

And what could better help a developer gain a "big picture" perspective than a big picture? To help you understand how the Subversion libraries fit together, see Figure 7-1,a diagram of Subversion's layers. Program flow begins at the top of the diagram (initiated by the user) and flows "downward".

Figure 7.1. Subversion's "Big Picture"



Another benefit of modularity is the ability to replace a given module with a whole new library that implements the same API without affecting the rest of the code base. In some sense, this happens within Subversion already. The libsvn_ra_dav and libsvn_ra_local libraries both implement the same interface, and both communicate with the Repository Layer, except that the former uses a network to talk to that layer, and the latter talks to it directly.

The client itself also highlights modularity in the Subversion design. While Subversion currently comes with only a command−line client program, there are already a few other programs being developed by third parties to act as GUIs for Subversion. Again, these GUIs use the same APIs that the stock command−line client does. Subversion's libsvn_client library is the one−stop shop for most of the functionality necessary for designing a working Subversion client (see the section called ˆClient Layer˜).

Repository Layer

When referring to Subversion's Repository Layer, we're generally talking about two libraries˙ the repository library, and the filesystem library. These libraries provide the storage and reporting mechanisms for the various revisions of your version−controlled data. This layer is connected to the Client Layer via the Repository Access Layer, and is, from the perspective of the Subversion user, the stuff at the "other end of the line."

The Subversion Filesystem is accessed via the libsvn_fs API, and is not a kernel−level filesystem that one would install in an operating system (like the Linux ext2 or NTFS), but a virtual filesystem. Rather than storing "files" and "directories" as real files and directories (as in, the kind you can navigate through using your favorite shell program), it uses a database system for its back−end storage mechanism. Currently, the database system in use is Berkeley DB. 6 However, there has been considerable interest by the development community in giving future releases of Subversion the ability to use other back−end database systems, perhaps through a mechanism such as Open Database Connectivity (ODBC).

The filesystem API exported by libsvn_fs contains the kinds of functionality you would expect from any other filesystem API: you can create and remove files and directories, copy and move them around, modify file contents, and so on. It also has features that are not quite as common, such as the ability to add, modify, and remove metadata ("properties") on each file or directory. Furthermore, the Subversion Filesystem is a versioning filesystem, which means that as you make changes to your directory tree, Subversion remembers what your tree looked like before those changes. And before the previous changes. And the previous ones. And so on, all the way back through versioning time to (and just beyond) the moment you first started adding things to the filesystem.

All the modifications you make to your tree are done within the context of a Subversion transaction. The following is a simplified general routine for modifying your filesystem:

1. Begin a Subversion transaction.
2. Make your changes (adds, deletes, property modifications, etc.).
3. Commit your transaction.

Once you have committed your transaction, your filesystem modifications are permanently stored as historical artifacts. Each of these cycles generates a single new revision of your tree, and each revision is forever accessible as an immutable snapshot of "the way things were."

The Transaction Distraction

The notion of a Subversion transaction, especially given its close proximity the database code in libsvn_fs, can become easily confused with the transaction support provided by the underlying database itself. Both types of transaction exist to provide atomicity and isolation. In other words, transactions give you the ability to perform a set of actions in an "all or nothing" fashion˙ either all the actions in the set complete with success, or they all get treated as if *none* of them ever happened˙ and in a way that does not interfere with other processes acting on the data.

Database transactions generally encompass small operations related specifically to the modification of data in the database itself (such as changing the contents of a table row). Subversion transactions are larger in scope, encompassing higher−level operations like making modifications to a set of files and directories which are intended to be stored as the next revision of the filesystem tree. If that isn't confusing enough, consider this: Subversion uses a database transaction during the creation of a Subversion transaction (so that if the creation of Subversion transaction fails, the database will look as if we had never attempted that creation in the first place)!

Fortunately for users of the filesystem API, the transaction support provided by the database system itself is hidden almost entirely from view (as should be expected from a properly modularized library scheme). It is only when you start digging into the implementation of the filesystem itself that such things become visible (or interesting).

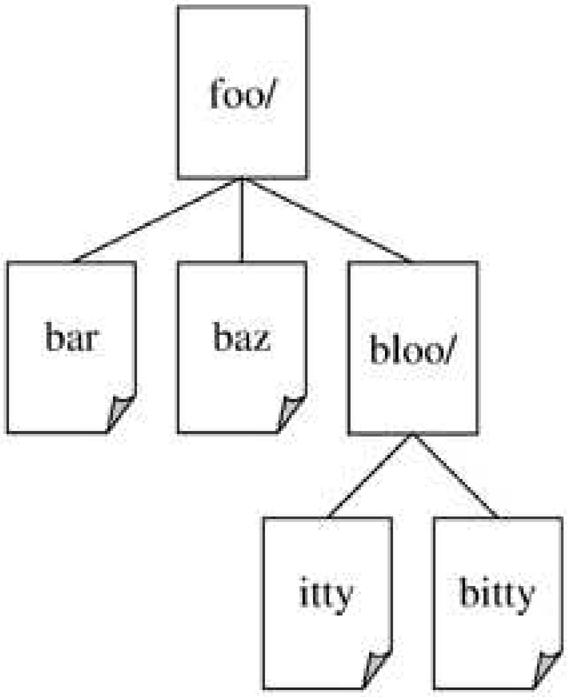
Most of the functionality provided by the filesystem interface comes as an action that occurs on a filesystem path. That is, from outside of the filesystem, the primary mechanism for describing and accessing the individual revisions of files and directories comes through the use of path strings like `/foo/bar`, just as if you were addressing files and

6The choice of Berkeley DB brought several automatic features that Subversion needed, such as data integrity, atomic writes, recoverability, and hot backups.

directories through your favorite shell program. You add new files and directories by passing their paths−to−be to the right API functions. You query for information about them by the same mechanism.

Unlike most filesystems, though, a path alone is not enough information to identify a file or directory in Subversion. Think of a directory tree as a two−dimensional system, where a node's siblings represent a sort of left−and−right motion, and descending into subdirectories a downward motion. Figure 7−2 shows a typical representation of a tree as exactly that.

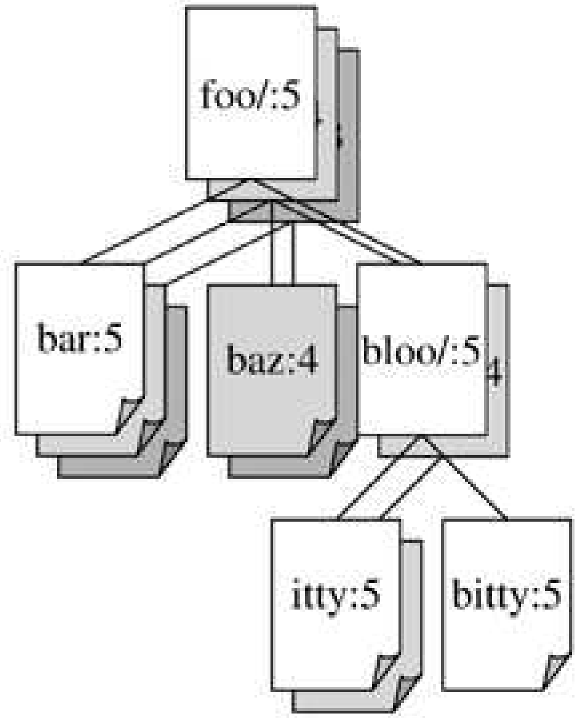
Figure 7.2. Files and directories in two dimensions



Of course, the Subversion filesystem has a nifty third dimension that most filesystems do not have˙ Time! [7] In the filesystem interface, nearly every function that has a *path* argument also expects a *root* argument. This svn_fs_root_t argument describes either a revision or a Subversion transaction (which is usually just a revision−to−be), and provides that third−dimensional context needed to understand the difference between /foo/bar in revision 32, and the same path as it exists in revision 98. Figure 7−3 shows revision history as an added dimension to the Subversion filesystem universe.

Figure 7.3. Revisioning Time˙ the third dimension!

7 We understand that this may come as a shock to sci−fi fans who have long been under the impression that Time was actually the *fourth* dimension, and we apologize for any emotional trauma induced by our assertion of a different theory.



As we mentioned earlier, the libsvn_fs API looks and feels like any other filesystem, except that it has this wonderful versioning capability. It was designed to be usable by any program interested in a versioning filesystem. Not coincidentally, Subversion itself is interested in that functionality. But while the filesystem API should be sufficient for basic file and directory versioning support, Subversion wants more˙ and that is where libsvn_repos comes in.

The Subversion repository library (libsvn_repos) is basically a wrapper library around the filesystem functionality. This library is responsible for creating the repository layout, making sure that the underlying filesystem is initialized, and so on. Libsvn_repos also implements a set of hooks˙ scripts that are executed by the repository code when certain actions take place. These scripts are useful for notification, authorization, or whatever purposes the repository administrator desires. This type of functionality, and other utility provided by the repository library, is not strictly related to implementing a versioning filesystem, which is why it was placed into its own library.

Developers who wish to use the libsvn_repos API will find that it is not a complete wrapper around the filesystem interface. That is, only certain major events in the general cycle of filesystem activity are wrapped by the repository interface. Some of these include the creation and commit of Subversion transactions, and the modification of revision properties. These particular events are wrapped by the repository layer because they have hooks associated with them. In the future, other events may be wrapped by the repository API. All of the remaining filesystem interaction will continue to occur directly with libsvn_fs API, though.

For example, here is a code segment that illustrates the use of both the repository and filesystem interfaces to create a new revision of the filesystem in which a directory is added. Note that in this example (and all others throughout this book), the `SVN_ERR` macro simply checks for a non−successfulerror return from the function it wraps, and returns that error if it exists.

Example 7.1. Using the Repository Layer

```
/* Create a new directory at the path NEW_DIRECTORY in the Subversion
   repository located at REPOS_PATH.  Perform all memory allocation in
```

```
   POOL.  This function will create a new revision for the addition of
   NEW_DIRECTORY.  */
static svn_error_t *
make_new_directory (const char *repos_path,
                    const char *new_directory,
                    apr_pool_t *pool)
{
  svn_error_t *err;
  svn_repos_t *repos;
  svn_fs_t *fs;
  svn_revnum_t youngest_rev;
  svn_fs_txn_t *txn;
  svn_fs_root_t *txn_root;
  const char *conflict_str;

  /* Open the repository located at REPOS_PATH.  */
  SVN_ERR (svn_repos_open (&repos, repos_path, pool));

  /* Get a pointer to the filesystem object that is stored in
     REPOS.  */
  fs = svn_repos_fs (repos);

  /* Ask the filesystem to tell us the youngest revision that
     currently exists.  */
  SVN_ERR (svn_fs_youngest_rev (&youngest_rev, fs, pool));

  /* Begin a new transaction that is based on YOUNGEST_REV.  We are
     less likely to have our later commit rejected as conflicting if we
     always try to make our changes against a copy of the latest snapshot
     of the filesystem tree.  */
  SVN_ERR (svn_fs_begin_txn (&txn, fs, youngest_rev, pool));

  /* Now that we have started a new Subversion transaction, get a root
     object that represents that transaction.  */
  SVN_ERR (svn_fs_txn_root (&txn_root, txn, pool));

  /* Create our new directory under the transaction root, at the path
     NEW_DIRECTORY.  */
  SVN_ERR (svn_fs_make_dir (txn_root, new_directory, pool));

  /* Commit the transaction, creating a new revision of the filesystem
     which includes our added directory path.  */
  err = svn_repos_fs_commit_txn (&conflict_str, repos,
                                 &youngest_rev, txn);
  if (err == SVN_NO_ERROR)
    {
      /* No error?  Excellent!  Print a brief report of our success.  */
      printf ("Directory '%s' was successfully added as new revision "
              "'%" SVN_REVNUM_T_FMT "'.\n", new_directory, youngest_rev);
    }
  else if (err->apr_err == SVN_ERR_FS_CONFLICT))
    {
      /* Uh-oh.  Our commit failed as the result of a conflict
         (someone else seems to have made changes to the same area
         of the filesystem that we tried to modify).  Print an error
         message.  */
      printf ("A conflict occured at path '%s' while attempting "
              "to add directory '%s' to the repository at '%s'.\n",
              conflict_str, new_directory, repos_path);
    }
  else
    {
      /* Some other error has occurred.  Print an error message.  */
      printf ("An error occured while attempting to add directory '%s' "
              "to the repository at '%s'.\n",
              new_directory, repos_path);
    }

  /* "Disconnect" from the repository.  */
  (void) svn_repos_close (repos);

  /* Return the result of the attempted commit to our caller.  */
```

```
  return err;
}
```

As you can see in the previous code segment, calls were made to both the repository and filesystem interfaces. Note that we could just as easily have committed the transaction using `svn_fs_commit_txn`. But the filesystem API knows nothing about the repository library's hook mechanism. If you want your Subversion repository to automatically perform some set of non−Subversion tasks every time you commit a transaction (like, for example, sending an email that describes all the changes made in that transaction to your developer mailing list), you need to use the libsvn_repos−wrapped version of that function˙ `svn_repos_fs_commit_txn`. This function will actually first run the "pre−commit" hook script if one exists, then commit the transaction, and finally will run a "post−commit" hook script. The hooks provide a special kind of reporting mechanism that does not really belong in the core filesystem library itself.

The hook mechanism requirement is but one of the reasons for the abstraction of a separate repository library from the rest of the filesystem code. The libsvn_repos API provides several other important utilities to Subversion. These include the abilities to:

1. create, open, destroy, and perform recovery steps on a Subversion repository and the filesystem included in that repository.
2. describe the differences between two filesystem trees.
3. query for the commit log messages associated with all (or some) of the revisions in which a set of files was modified in the filesystem.
4. generate a human−readable "dump" of the filesystem, a complete representation of the revisions in the filesystem.
5. parse that dump format, loading the dumped revisions into a different Subversion repository.

As Subversion continues to evolve, the repository library will grow with the filesystem library to offer increased functionality and configurable option support.

Repository Access Layer

If the Subversion Repository Layer is at "the other end of the line", the Repository Access Layer is the line itself. Charged with marshalling data between the client libraries and the repository, this layer includes the libsvn_ra module loader library, the RA modules themselves (which currently includes libsvn_ra_local and libsvn_ra_dav), and any additional libraries needed by one or more of those RA modules, such as the mod_dav_svn Apache module with which libsvn_ra_dav communicates.

Since Subversion uses URLs to identify its repository resources, the protocol portion of the URL schema (usually `http:`, `https:`, or `file:`) is used to determine which RA module will handle the communications. Each module registers a list of the protocols it knows how to "speak" so that the RA loader can, at runtime, determine which module to use for the task at hand. You can determine which RA modules are available to the Subversion command−line client, and what protocols they claim to support, by running **svn −−version**In the following example, both of the RA modules that are included with Subversion have been successfully located by the client program.

```
$ svn --version
svn, version 0.14.3 (dev build)
   compiled Oct  7 2002, 07:52:08

Copyright (C) 2000-2002 CollabNet.
Subversion is open source software, see http://subversion.tigris.org/

The following repository access (RA) modules are available:

* ra_dav : Module for accessing a repository via WebDAV (DeltaV) protocol.
```

```
  - handles 'http' schema
  - handles 'https' schema
* ra_local : Module for accessing a repository on local disk.
  - handles 'file' schema
```

For the majority of this section we will be highlighting libsvn_ra_dav, which is the most commonly used RA module for real−worldrepository communications. Unlike libsvn_ra_local, which offers no networking capabilities at all, libsvn_ra_dav is designed for use by clients that are being run on different machines than the servers with which they communicating, specifically machines reached using URLs that contain the `http:` or `https:` protocol portions. To understand how this module works, we should first mention a couple of other key components in this particular configuration of the Repository Access Layer˙ the powerful Apache HTTP Server, and the Neon HTTP/WebDAV client library.

Subversion's official server is the Apache HTTP Server. Apache is a time−tested,extensible open−sourceserver process that is ready for serious use. It can sustain a high network load and runs on many platforms. The Apache server supports a number of different standard authentication protocols, and can be extended through the use of modules to support many others. It also supports optimizations like network pipelining and caching. By using Apache as a server, Subversion gets all of these features for free. And since most firewalls already allow HTTP traffic to pass through, sysadmins typically don't even have to change their firewall configurations to allow Subversion to work.

Subversion uses WebDAV (with DeltaV) as its network protocol. You can read more about this in the WebDAV section of this chapter, but in short, WebDAV and DeltaV are extensions to the standard HTTP 1.1 protocol that enable sharing and versioning of files over the web. Apache 2.0 comes with mod_dav, an Apache module that understands the DAV extensions to HTTP. Subversion itself supplies mod_dav_svn, though, which is another Apache module that works in conjunction with (really, as a back−endto) mod_dav to provide Subversion's specific implementations of WebDAV and DeltaV.

When communicating with a repository over HTTP, the RA loader library chooses libsvn_ra_dav as the proper access module. The Subversion client makes calls into the generic RA interface, and libsvn_ra_dav maps those calls (which embody rather large−scaleSubversion actions) to a set of HTTP/WebDAV requests. Using the Neon library, libsvn_ra_dav transmits those requests to the Apache server. Apache receives these requests (exactly as it does generic HTTP requests that your web browser might make), notices that the requests are directed at a URL that is configured as a DAV location (using the `Location` directive in `httpd.conf`), and hands the request off to its own mod_dav module. When properly configured, mod_dav knows to use Subversion's mod_dav_svn for any filesystem−related needs, as opposed to the generic mod_dav_fs that comes with Apache. So ultimately, the client is communicating with mod_dav_svn, which binds directly to the Subversion Repository Layer.

That was a simplified description of the actual exchanges taking place, though. For example, the Subversion repository might be protected by Apache's authorization directives. This could result in initial attempts to communicate with the repository being rejected by Apache on authorization grounds. At this point, libsvn_ra_dav gets back the notice from Apache that insufficient identification was supplied, and calls back into the Client Layer to get some updated authentication data. If the data is supplied correctly, and the user has the permissions that Apache seeks, libsvn_ra_dav's next automatic attempt at performing the original operation will be granted, and all will be well. If sufficient authentication information cannot be supplied, the request will ultimately fail, and the client will report the failure to the user.

By using Neon and Apache, Subversion gets free functionality in several other complex areas, too. For example, if Neon finds the OpenSSL libraries, it allows the Subversion client to attempt to use SSL−encryptedcommunications with the Apache server (whose own mod_ssl can "speak the language"). Also, both Neon itself and Apache's mod_deflate can understand the "deflate" algorithm (the same used by the PKZIP and gzip programs), so requests can be sent in smaller, compressed chunks across the wire. Other complex features that Subversion hopes to support in the future include the ability to automatically handle server−specifiedredirects (for example, when a repository has been moved to a new canonical URL) and taking advantage of HTTP pipelining.

Of course, not all communications with a Subversion repository require a powerhouse server process and a network layer. For users who simply wish to access the repositories on their local disk, they may do so using `file:` URLs and the functionality provided by libsvn_ra_local. This RA module binds directly with the repository and filesystem libraries, so no network communication is required at all.

And for those who wish to access a Subversion repository using still another protocol˙ well, that is precisely why the Repository Access Layer is modularized! Developers can simply write a new library that implements the RA interface on one side and communicates with the repository on the other. Your new protocol could use straight socket connections (bypassing the likes of Apache), or inter-process communication (IPC) calls, or˙ let's get crazy, shall we?˙ you could even implement an email-based protocol. Subversion supplies the APIs; you supply the creativity.

Client Layer

On the client side, the Subversion working copy is where all the action takes place. The bulk of functionality implemented by the client-side libraries exists for the sole purpose of managing working copies˙ directories full of files and other subdirectories which serve as a sort of local, editable "reflection" of one or more repository locations˙ and propogating changes to and from the Repository Access layer.

Subversion's working copy library, libsvn_wc, is directly responsible for managing the data in the working copies. To accomplish this, the library stores administrative information about each working copy directory within a special subdirectory. This subdirectory, named `.svn` is present in each working copy directory and contains various other files and directories which record state and provide a private workspace for administrative action. For those familiar with CVS, this `.svn` subdirectory is similar in purpose to the `CVS` administrative directories found in CVS working copies. For more information about the `.svn` administrative area, see the section called ˆInside the Working Copy Administration Area˜in this chapter.

The Subversion client library, libsvn_client, has the broadest responsibility; its job is to mingle the functionality of the working copy library with that of the Repository Access Layer, and then to provide the highest-level API to any application that wishes to perform general revision control actions. For example, the function `svn_client_checkout` takes a URL as an argument. It passes this URL to the RA layer and opens an authenticated session with a particular repository. It then asks the repository for a certain tree, and sends this tree into the working copy library, which then writes a full working copy to disk (`.svn` directories and all).

The client library is designed to be used by any application. While the Subversion source code includes a standard command-line client, it should be very easy to write any number of GUI clients on top of the client library. New GUIs (or any new client, really) for Subversion need not be clunky wrappers around the included command-line client˙ they have full access via the libsvn_client API to same functionality, data, and callback mechanisms that the command-line client uses.

Binding Directly˙ A Word About Correctness

Why should your GUI program bind directly with a libsvn_client instead of acting as a wrapper around a command-line program? Besides simply being more efficient, this can address potential correctness issues as well. A command-line program (like the one supplied with Subversion) that binds to the client library needs to effectively translate feedback and requested data bits from C types to some form of human-readable output. This type of translation can be lossy. That is, the program may not display all of the information harvested from the API, or may combine bits of information for compact representation.

If you wrap such a command-line program with yet another program, the second program has access only to already-interpreted (and as we mentioned, likely incomplete) information, which it must *again* translate into *its* representation format. With each layer of wrapping, the integrity of the original data is potentially tainted more and more, much like the result of making a copy of a copy (of a copy ...) of a favorite audio or video cassette.

Using the APIs

Developing applications against the Subversion library APIs is fairly straightforward. All of the public header files live in the `subversion/include` directory of the source tree. These headers are copied into your system locations when you build and install Subversion itself from source. These headers represent the entirety of the functions and types meant to be accessible by users of the Subversion libraries.

The first thing you might notice is that Subversion's datatypes and functions are namespace protected. Every public Subversion symbol name begins with `"svn_"`, followed by a short code for the library in which the symbol is defined (such as "wc", "client", "fs", etc.), followed by a single underscore ("_") and then the rest of the symbol name. Semi−public functions (used among source files of a given library but not by code outside that library, and found inside the library directories themselves) differ from this naming scheme in that instead of a single underscore after the library code, they use a double underscore ("__"). Functions private a given source file have no special prefixing, and are declared `static`. Of course, a compiler isn't iterested in these naming conventions, but they defi− nitely help to clarify the scopy of a given function or datatype.

The Apache Portable Runtime Library

Along with Subversion's own datatype, you will see many references to datatypes that begin with "apr_"˙ symbols from the Apache Portable Runtime (APR) library. APR is Apache's portability library, originally carved out of its server code as an attempt to separate the OS−specific bits from the OS−independent portions of the code. The result was a library that provides a generic API for performing operations that differ mildly˙ or wildly˙ from OS to OS. While Apache HTTP Server was obviously the first user of the APR library, the Subversion developers immediately recognized the value of using APR as well. This means that there are practically no OS−specific code portions in Subversion itself. Also, it means that the Subversion client compiles and runs anywhere that the server does. Cur− rently this list includes all flavors of Unix, Win32, BeOS, OS/2, and Mac OS X.

In addition to providing consistent implementations of system calls that differ across operating systems, [8] APR gives Subversion immediate access to many custom datatypes, such as dynamic arrays and hash tables. Subversion uses these types extensively throughout the codebase. But perhaps the most pervasive APR datatype, found in nearly every Subversion API prototype, is the apr_pool_t˙ the APR memory pool. Subversion uses pools internally for all its memory allocation needs, and while a person coding against the Subversion APIs is not required to do the same, they are required to provide pools to the API functions that need them. This means that users of the Subversion API must also link against APR, must call `apr_initialize()` to initialize the APR subsystem, and then must acquire a pool for use with Subversion API calls. See the section called ˆProgramming with Memory Pools˜ for more infor− mation.

URL and Path Requirements

With remote version control operation as the whole point of Subversion's existence, it makes sense that some atten− tion has been paid to internationalization (i18n) support. After all, while "remote" might mean "across the office", it could just as well mean "across the globe." To facilitate this, all of Subversion's public interfaces that accept path ar− guments expect those paths to be canonicalized, and encoded in UTF−8. This means, for example, that any new client binary that drives the libsvn_client interface needs to first convert paths from the locale−specific encoding to UTF−8 before passing those paths to the Subversion libraries, and then re−convert any resultant output paths from Subversion back into the locale's encoding before using those paths for non−Subversion purposes. Fortunately, Sub− version provides a suite of functions (see `subversion/include/svn_utf.h`) that can be used by any program to do these conversions.

Also, Subversion APIs require all URL parameters to be properly URI−encoded. So, instead of passing `file:///home/username/My File.txt` as the URL of a file named `My File.txt`, you need to pass `file:///home/username/My%20File.txt`. Again, Subversion supplies helper functions that your application can use˙ `svn_path_uri_encode` and `svn_path_uri_decode`, for URI encoding and decoding, respectively.

Using Languages Other than C and C++

If you are interested in using the Subversion libraries in conjunction with something other than a C program˙ say a Python script or Java application˙ Subversion has some initial support for this via the Simplified Wrapper and In− terface Generator (SWIG). The SWIG bindings for Subversion are located in `subversion/bindings/swig` and are slowly maturing into a usable state. These bindings allow you to call Subversion API functions indirectly, using wrappers that translate the datatypes native to your scripting language into the datatypes needed by Subversion's C libraries.

There is an obvious benefit to accessing the Subversion APIs via a language binding˙ simplicity. Generally speak− ing, languages such as Python and Perl are much more flexible and easy to use than C or C++. The sort of high−level datatypes and context−driven typechecking provided by these languages are often better at handling information that

[8] Subversion uses ANSI system calls and datatypes as much as possible.

comes from users. As you know, only a human can botch up the input to a program as well as they do, and the scripting−typelanguage simply handle that misinformation more gracefully. Of course, often that flexibility comes at the cost of performance. That is why using a tightly−optimized,C−basedinterface and library suite, combined with a powerful, flexible binding language is so appealing.

Let's look at an example that uses Subversion's Python SWIG bindings. Our example will do the same thing as our last example. Note the difference in size and complexity of the function this time!

Example 7.2. Using the Repository Layer with Python

```
from svn import fs
import os.path

def crawl_filesystem_dir (root, directory, pool):
  """Recursively crawl DIRECTORY under ROOT in the filesystem, and return
  a list of all the paths at or below DIRECTORY.  Use POOL for all
  allocations."""

  # Get the directory entries for DIRECTORY.
  entries = fs.dir_entries(root, directory, pool)

  # Initialize our returned list with the directory path itself.
  paths = [directory]

  # Loop over the entries
  names = entries.keys()
  for name in names:
    # Calculate the entry's full path.
    full_path = os.path.join(basepath, name)

    # If the entry is a directory, recurse.  The recursion will return
    # a list with the entry and all its children, which we will add to
    # our running list of paths.
    if fs.is_dir(fsroot, full_path, pool):
      subpaths = crawl_filesystem_dir(root, full_path, pool)
      paths.extend(subpaths)

    # Else, it is a file, so add the entry's full path to the FILES list.
    else:
      paths.append(full_path)

  return paths
```

An implementation in C of the previous example would stretch on quite a bit longer. The same routine in C would need to pay close attention to memory usage, and need to use custom datatypes for representing the hash of entries and the list of paths. Python has hashes and lists (called "dictionaries" and "sequences", respectively) as built−in datatypes, and provides a wonderful selection of methods for operating on those types. And since Python uses refer−ence counting and garbage collection, users of the language don't have to bother themselves with allocating and free−ing memory.

In the previous section of this chapter, we mentioned the `libsvn_client` interface, and how it exists for the sole purpose of simplifying the process of writing a Subversion client. The following is a brief example of how that li−brary can be accessed via the SWIG bindings. In just a few lines of Python, you can check out a fully functional Subversion working copy!

Example 7.3. A simple script to check out a working copy.

```
#!/usr/bin/env python
import sys
```

```
from svn import util, _util, _client

def usage():
  print "Usage: " + sys.argv[0] + " URL PATH\n"
  sys.exit(0)

def run(url, path):
  # Initialize APR and get a POOL.
  _util.apr_initialize()
  pool = util.svn_pool_create(None)

  # Checkout the HEAD of URL into PATH (silently)
  _client.svn_client_checkout(None, None, url, path, -1, 1, None, pool)

  # Cleanup our POOL, and shut down APR.
  util.svn_pool_destroy(pool)
  _util.apr_terminate()

if __name__ == '__main__':
  if len(sys.argv) != 3:
    usage()
  run(sys.argv[1], sys.argv[2])
```

Currently, it is Subversion's Python bindings that are the most complete. Some attention is also being given to the Java bindings. Once you have the SWIG interface files properly configured, generation of the specific wrappers for all the supported SWIG languages (which currently includes versions of Tcl, Python, Perl, Java, Ruby, Mzscheme, Guile, and PHP) should theoretically be trivial. Still, some extra programming is required to compensate for complex APIs that SWIG needs some help generalizing. For more information on SWIG itself, see the project's website at `http://www.swig.org`.

Inside the Working Copy Administration Area

As we mentioned earlier, each directory of a Subversion working copy contains a special subdirectory called `.svn` which houses administrative data about that working copy directory. Subversion uses the information in `.svn` to keep track of things like:

- Which repository location(s) are represented by the files and subdirectories in the working copy directory.
- What revision of each of those files and directories are currently present in the working copy.
- Any user−definedproperties that might be attached to those files and directories.
- Pristine (un−edited)copies of the working copy files.

While there are several other bits of data stored in the `.svn` directory, we will examine only a couple of the most important items.

The Entries File

Perhaps the single most important file in the `.svn` directory is the `entries` file. The entries file is an XML document which contains the bulk of the administrative information about a versioned resource in a working copy directory. It is this one file which tracks the repository URLs, pristine revision, file checksums, pristine text and property timestamps, scheduling and conflict state information, last−knowncommit information (author, revision, timestamp), local copy history˙ practically everything that a Subversion client is interested in knowing about a versioned (or to−be−versioned)resource!

Comparing the Administrative Areas of Subversion and CVS

A glance inside the typical `.svn` directory turns up a bit more than what CVS maintains in its `CVS` administrative di−

rectories. The `entries` file contains XML which describes the current state of the working copy directory, and basically serves the purposes of CVS's `Entries`, `Root`, and `Repository` files combined. Authentication data is also stored within `.svn`, rather than in a single .cvspass–likefile.

The following is an example of an actual entries file:

Example 7.4. Contents of a typical `.svn/entries` file

```
<?xml version="1.0" encoding="utf-8"?>
<wc-entries
   xmlns="svn:">
<entry
   committed-rev="1"
   name="svn:this_dir"
   committed-date="2002-09-24T17:12:44.064475Z"
   url="file:///home/cmpilato/tests/.greek-repo/A/D"
   kind="dir"
   revision="1"/>
<entry
   committed-rev="1"
   name="gamma"
   text-time="2002-09-26T21:09:02.000000Z"
   committed-date="2002-09-24T17:12:44.064475Z"
   checksum="QSE4vWd9ZM0cMvr7/+YkXQ=="
   kind="file"
   prop-time="2002-09-26T21:09:02.000000Z"/>
<entry
   name="zeta"
   kind="file"
   schedule="add"
   revision="0"/>
<entry
   url="file:///home/cmpilato/tests/.greek-repo/A/B/delta"
   name="delta"
   kind="file"
   schedule="add"
   revision="0"/>
<entry
   name="G"
   kind="dir"/>
<entry
   name="H"
   kind="dir"
   schedule="delete"/>
</wc-entries>
```

As you can see, the entries file is essentially a list of entries. Each `entry` tag represents one of three things: the working copy directory itself (noted by having its *`name`* attribute set to `"svn:this-dir"`), a file in that working copy directory (noted by having its *`kind`* attribute set to `"file"`), or a subdirectory in that working copy (*`kind`* here is set to `"dir"`). The files and subdirectories whose entries are stored in this file are either already under version control, or (as in the case of the file named `zeta` above) are scheduled to be added to version control when the user next commits this working copy directory's changes. Each entry has a unique name, and each entry has a node kind.

Developers should be aware of some special rules that Subversion uses when reading and writing its `entries` files. While each entry has a revision and URL associated with it, note that not every `entry` tag in the sample file has explicit *`revision`* or *`url`* attributes attached to it. Subversion allows entries to not explicitly store those two attributes when their values are the same as (in the *`revision`* case) or trivially calculable from [9] (in the *`url`* case) the data stored in the `"svn:this-dir"` entry. Note also that for subdirectory entries, Subversion stores only the crucial attributes˙ name, kind, url, revision, and schedule. In an effort to reduce duplicated information, Subversion dictates

9 That is, the URL for the entry is the same as the concatenation of the parent directory's URL and the entry's name.

that the method for determining the full set of information about a subdirectory is to traverse down into that subdirectory, and read the "svn:this-dir" entry from its own .svn/entries file. However, a reference to the subdirectory is kept in its parent's entries file, with enough information to permit basic versioning operations in the event that the subdirectory itself is actually missing from disk.

Pristine Copies and Property Files

As mentioned before, the .svn directory also holds the pristine "text-base" versions of files. Those can be found in .svn/text-base. The benefits of these pristine copies are multiple — network-free checks for local modifications and "diff" reporting, network-free reversion of modified or missing files, smaller transmission of changes to the server — but comes at the cost of having each versioned file stored at least twice on disk. These days, this seems to be a negligible penalty for most files. However, the situation gets uglier as the size of your versioned files grows. Some attention is being given to making the presence of the "text-base" an option. Ironically though, it is as your versioned files' sizes get larger that the existence of the "text-base" becomes more crucial — who wants to transmit a huge file across a network just because they want to commit a tiny change to it?

Similar in purpose to the "text-base" files are the property files and their pristine "prop-base" copies, located in .svn/props and .svn/prop-base respectively. Since directories can have properties, too, there are also .svn/dir-props and .svn/dir-prop-base files. Each of these property files ("working" and "base" versions) uses a simple "hash-on-disk" file format for storing the property names and values.

The Authentication Area

We will wrap up our peek at the working copy internals by noting the .svn/auth directory. Here is where Subversion stores a cache of various data used when authenticating against a server that requires such. Subversion uses OS-level permissioning to secure the contents of these files, but also gives users the option of disabling this cache altogether (via a per-use command-line argument, or more conveniently via the user's configuration files).

WebDAV

WebDAV ("Web-based Distributed Authoring and Versioning") is an extension of the standard HTTP protocol designed to make the web into a read/write medium, instead of the basically read-only medium that exists today. The theory is that directories and files can be shared — as both readable and writable objects — over the web. RFC 2518 describes the WebDAV extensions to HTTP, and is available (along with a lot of other useful information) at http://www.webdav.org/.

A number of operating system file browsers are already able to mount networked directories using WebDAV. On Win32, the Windows Explorer can browse what it calls "WebFolders" (which are just WebDAV-ready network locations) as if they were regular folders on the local machine. Mac OS X also has this capability, as does the Nautilus browser for GNOME.

Browsing WebDAV locations using a standard file browser

WebDAV-enabled servers publish their resources as collections and the files (and other collections) within them. Today's operation systems are becoming extremely Web-aware, and this concept maps so easily to the notion of directories and files that many operation systems have built-in support for viewing those locations as regular folders. And these folders can be browsed just like system folders — you can open and copy files from the WebDAV folders as if they were sitting on your local drive.

For example, on recent versions of Windows, one of the items that appears when you open My Computer is Web Folders. Opening that icon will show you list of registered WebDAV locations that you can browse further into, as well as an icon for adding a new Web Folder. In fact, if you'd like to see how this works, open the Add Web Folder icon, and register the URL http://svn.collab.net/repos/svn/trunk. When you've finished, you'll have an icon in your Web Folders window that, if opened, will connect you directly to the head of Subversion's own development tree. Imagine that — bleeding edge Subversion code that you can copy and paste right off the server and onto your local drive!

However, RFC 2518 doesn't fully implement the "versioning" aspect of WebDAV. A separate committee has created RFC 3253, known as the DeltaV extensions to WebDAV, available at `http://www.webdav.org/deltav/`. These extensions add version−controlconcepts to WebDAV, and ultimately to HTTP.

So how does all of this apply to Subversion? Subversion uses HTTP, extended by WebDAV and DeltaV, as its primary network protocol. Rather than implementing a new proprietary protocol, the Subversion developers decided to simply map the versioning concepts and actions used by Subversion onto the concepts exposed by RFCs 2518 and 3253.

It is important to understand that while Subversion uses DeltaV for communication, the Subversion client is *not* a general−purpose DeltaV client. In fact, it expects some custom features from the server. Further, the Subversion server is not a general−purposeDeltaV server. It only implements a strict subset of the DeltaV specification. A WebDAV or DeltaV client may very well be able to interoperate with it, but only if that client operates within the narrow confines of those features that the server has implemented. Future versions of Subversion will more completely address WebDAV interoperability.

At the moment, most DAV browsers and clients do not yet support DeltaV; this means that a Subversion repository can be viewed or mounted only as a read−onlyresource. And on the flip side, a Subversion client cannot checkout a working copy from a generic WebDAV server; it expects a specific subset of DeltaV features.

Programming with Memory Pools

Almost every developer who has used the C programming language has at some point sighed at the daunting task of managing memory usage. Allocating enough memory to use, keeping track of those allocations, freeing the memory when you no longer need it˙ these tasks can be quite complex. And of course, failure to do those things properly can result in a program that crashes itself, or worse, crashes the computer. Fortunately, the APR library that Subversion depends on for portability provides the apr_pool_t type, which represents a "pool" of memory.

A memory pool is an abstract representation of a chunk of memory allocated for use by a program. Rather than requesting memory directly from the OS using the standard `alloc()` and friends, programs that link against APR can simply request that a pool of memory be created (using the `apr_pool_create()` function). APR will allocate a moderately sized chunk of memory from the OS, and that memory will be instantly available for use by the program. Any time the program needs some of the pool memory, it uses one of the APR pool API functions, like `apr_palloc()`, which returns a generic memory location from the pool. The program can keep requesting bits and pieces of memory from the pool, and APR will keep granting the requests. Pools will automatically grow in size to accommodate programs that request more memory than the original pool contained, until of course there is no more memory available on the system.

Now, if this were the end of the pool story, it would hardly have merited special attention. Fortunately, that's not the case. Pools can not only be created; they can also be cleared and destroyed, using `apr_pool_clear()` and `apr_pool_destroy()` respectively. This gives developers the flexibility to allocate several˙ or several thousand˙ things from the pool, and then clean up all of that memory with a single function call! Further, pools have hierarchy. You can make "subpools" of any previously created pool. When you clear a pool, all of its subpools are destroyed; if you destroy a pool, it and its subpools are destroyed.

Before we go further, developers should be aware that they probably will not find many calls to the APR pool functions we just mentioned in the Subversion source code. APR pools offer some extensibility mechanisms, like the ability to have custom "user data" attached to the pool, and mechanisms for registering cleanup functions that get called when the pool is destroyed. Subversion makes use of these extensions in a somewhat non−trivial way. So, Subversion supplies (and most of its code uses) the wrapper functions `svn_pool_create()`, `svn_pool_clear()`, and `svn_pool_destroy()`.

While pools are helpful for basic memory management, the pool construct really shines in looping and recursive scenarios. Since loops are often unbounded in their iterations, and recursions in their depth, memory consumption in these areas of the code can become unpredictable. Fortunately, using nested memory pools can be a great way to easily manage these potentially hairy situations. The following example demonstrates the basic use of nested pools in a situation that is fairly common˙ recursively crawling a directory tree, doing some task to each thing in the tree.

Example 7.5. Effective pool usage

```
/* Recursively crawl over DIRECTORY, adding the paths of all its file
   children to the FILES array, and doing some task to each path
   encountered.  Use POOL for the all temporary allocations, and store
   the hash paths in the same pool as the hash itself is allocated in.  */
static apr_status_t
crawl_dir (apr_array_header_t *files,
           const char *directory,
           apr_pool_t *pool)
{
  apr_pool_t *hash_pool = files->pool;  /* array pool */
  apr_pool_t *subpool = svn_pool_create (pool);  /* iteration pool */
  apr_dir_t *dir;
  apr_finfo_t finfo;
  apr_status_t apr_err;
  apr_int32_t flags = APR_FINFO_TYPE | APR_FINFO_NAME;

  apr_err = apr_dir_open (&dir, directory, pool);
  if (apr_err)
    return apr_err;

  /* Loop over the directory entries, clearing the subpool at the top of
     each iteration.  */
  for (apr_err = apr_dir_read (&finfo, flags, dir);
       apr_err == APR_SUCCESS;
       apr_err = apr_dir_read (&finfo, flags, dir))
    {
      const char *child_path;

      /* Skip entries for "this dir" ('.') and its parent ('..').  */
      if (finfo.filetype == APR_DIR)
        {
          if (finfo.name[0] == '.'
              && (finfo.name[1] == '\0'
                  || (finfo.name[1] == '.' && finfo.name[2] == '\0')))
            continue;
        }

      /* Build CHILD_PATH from DIRECTORY and FINFO.name.  */
      child_path = svn_path_join (directory, finfo.name, subpool);

      /* Do some task to this encountered path. */
      do_some_task (child_path, subpool);

      /* Handle subdirectories by recursing into them, passing SUBPOOL
         as the pool for temporary allocations.  */
      if (finfo.filetype == APR_DIR)
        {
          apr_err = crawl_dir (files, child_path, subpool);
          if (apr_err)
            return apr_err;
        }

      /* Handle files by adding their paths to the FILES array.  */
      else if (finfo.filetype == APR_REG)
        {
          /* Copy the file's path into the FILES array's pool.  */
          child_path = apr_pstrdup (hashpool, child_path);

          /* Add the path to the array.  */
          (*((const char **) apr_array_push (files))) = child_path;
        }

      /* Clear the per-iteration SUBPOOL.  */
      svn_pool_clear (subpool);
    }
```

```
  /* Check that the loop exited cleanly. */
  if (apr_err)
    return apr_err;

  /* Yes, it exited cleanly, so close the dir. */
  apr_err = apr_dir_close (dir);
  if (apr_err)
    return apr_err;

  /* Destroy SUBPOOL.  */
  svn_pool_destroy (subpool);

  return APR_SUCCESS;
}
```

The previous example demonstrates effective pool usage in *both* looping and recursive situations. Each recursion begins by making a subpool of the pool passed to the function. This subpool is used for the looping region, and cleared with each iteration. The result is memory usage is roughly proportional to the depth of the recursion, not to total number of file and directories present as children of the top–leveldirectory. When the first call to this recursive function finally finishes, there is actually very little data stored in the pool that was passed to it. Now imagine the extra complexity that would be present if this function had to `alloc()` and `free()` every single piece of data used!

Pools might not be ideal for every application, but they are extremely useful in Subversion. As a Subversion developer, you'llneed to grow comfortable with pools and how to weild them correctly. Memory usage bugs and bloating can be difficult to diagnose and fix regardless of the API, but the pool construct provided by APR has proven a tremendously convenient, time–savingbit of functionality.

Contributing to Subversion

The official source of information about the Subversion project is, of course, the project's website at `http://subversion.tigris.org`. There you can find information about getting access to the source code and participating on the discussion lists. The Subversion community always welcomes new members. If you are interested in participating in this community by contributing changes to the source code, here are some hints on how to get started.

Join the Community

The first step in community participation is to find a way to stay on top of the latest happenings. To do this most effectively, you will want to subscribe to the main developer discussion list (`<dev@subversion.tigris.org>`) and commit mail list (`<svn@subversion.tigris.org>`). By following these lists even loosely, you will have access to important design discussions, be able to see actual changes to Subversion source code as they occur, and be able to witness peer reviews of those changes and proposed changes. These e–mailbased discussion lists are the primary communication media for Subversion development. See the Mailing Lists section of the website for other Subversion–relatedlists you might be interested in.

But how do you know what needs to be done? It is quite common for a programmer to have the greatest intentions of helping out with the development, yet be unable to find a good starting point. After all, not many folks come to the community having already decided on a particular itch they would like to scratch. But by watching the developer discussion lists, you might see mentions of existing bugs or feature requests fly by that particularly interest you. Also, a great place to look for outstanding, unclaimed tasks is the Issue Tracking database on the Subversion website. There you will find the current list of known bugs and feature requests. If you want to start with something small, look for issues marked as "bite–sized'.

Get the Source Code

To edit the code, you need to have the code. This means you need to check out a working copy from the public Subversion source repository. As straightforward as that might sound, the task can be slightly tricky. Because Subversion's source code is versioned using Subversion itself, you actually need to "bootstrap" by getting a working Subversion client via some other method. The most common methods inclue downloading the latest binary distribution (if such is available for your platform), or downloading the latest source tarball and building your own Subversion client. If you build from source, make sure read the INSTALL file in the top level of the source tree for instructions.

After you have a working Subversion client, you are now poised to checkout a working copy of the Subversion source repository from `http://svn.collab.net/repos/svn/trunk`: [10]

```
$ svn checkout http://svn.collab.net/repos/svn/trunk subversion
A  HACKING
A  INSTALL
A  README
A  autogen.sh
A  build.conf
...
```

The above command will checkout the bleeding–edge,latest version of the Subversion source code into a subdirectory named `subversion` in your current working directory. Obviously, you can adjust that last argument as you see fit. Regardless of what you call the new working copy directory, though, after this operation completes, you will now have the Subversion source code. Of course, you will still need to fetch a few helper libraries (apr, apr–util, etc.)˙ see the INSTALL file in the top level of the working copy for details.

Become Familiar with Community Policies

Now that you have a working copy containing the latest Subversion source code, you will most certainly want to take a cruise through the HACKING file in that working copy's top–level directory. The HACKING file contains general instructions for contributing to Subversion, including how to properly format your source code for consistency with the rest of the codebase, how to describe your proposed changes with an effective change log message, how to test your changes, and so on. Commit priveleges on the Subversion source repository are earned˙ a government by meritocracy. [11] The HACKING file is an invaluable resource when it comes to making sure that your proposed changes earn the praises they deserve without being rejected on technicalities.

Make and Test Your Changes

With the code and community policy understanding in hand, you are ready to make your changes. It is best to try to make smaller but related sets of changes, even tackling larger tasks in stages, instead of making huge, sweeping modifications. Your proposed changes will be easier to understand (and therefore easier to review) if you disturb the fewest lines of code possible to accomplish your task properly. After making each set of proposed changes, your Subversion tree should be in a state in which the software compiles with no warnings.

Subversion has a fairly thorough [12] regression test suite, and your proposed changes are expected to not cause any of those tests to fail. By running **make check** (in Unix) from the top of the source tree, you can sanity–check your changes. The fastest way to get your code contributions rejected (other than failing to supply a good log message) is to submit changes that cause failure in the test suite.

In the best–case scenario, you will have actually added appropriate tests to that test suite which verify that your proposed changes actually work as expected. In fact, sometimes the best contribution a person can make is solely the addition of new tests. You can write regression tests for functionality that currently works in Subversion as a way to protect against future changes that might trigger failure in those areas. Also, you can write new tests that demonstrate known failures. For this purpose, the Subversion test suite allows you to specify that a given test is expected to fail (called an `XFAIL`), and so long as Subversion fails in the way that was expected, a test result of XFAIL itself is considered a success. Ultimately, the better the test suite, the less time wasted on diagnosing potentially obscure regression bugs.

Donate Your Changes

[10] Note that the URL checked out in the example above ends not with `svn`, but with a subdirectory thereof called `trunk`. See our discussion of Subversion's branching and tagging model for the reasoning behind this.

[11] While this may superficially appear as some sort of elitism, this "earn your commit priveleges" notion is about efficiency˙ whether it costs more in time and effort to review and apply someone else's changes that are likely to be safe and useful, versus the potential costs of undoing changes that are dangerous.

[12] You might want to grab some popcorn. "Thorough", in this instance, translates to somewhere around thirty minutes of non–interactive machine churn.

After making your modifications to the source code, compose a clear and concise log message to describe those changes and the reasons for them. Then, send an email to the developers list containing your log message and the output of **svn diff** (from the top of your Subversion working copy). If the community members consider your changes acceptable, someone who has commit priveleges (permission to make new revisions in the Subversion source repository) will add your changes to the public source code tree. Recall that permission to directly commit changes to the repository is granted on merit˙ if you demonstrate comprehension of Subversion, programming competency, and a "team spirit", you will likely be awarded that permission.

Chapter 8. Complete Reference

###TODO Write this

Appendix A. Subversion for CVS Users

This document is meant to be a quick−startguide for CVS users new to Subversion. It's not a substitute for real documentation and manuals; but it should give you a quick conceptual ˆdiff˜ when switching over.

The goal of Subversion is to take over the current and future CVS user base. Subversion not only includes new features, but attempts to fix certain ˆbroken˜ behaviors that CVS had. This means that you may be encouraged to break certain habits˙ ones that you forgot were odd to begin with.

Revision Numbers Are Different Now

In CVS, revision numbers are per−file. This is because CVS uses RCS as a backend; each file has a corresponding RCS file in the repository, and the repository is roughly laid out according to the structure of your project tree.

In Subversion, the repository looks like a single filesystem. Each commit results in an entirely new filesystem tree; in essence, the repository is an array of trees. Each of these trees is labeled with a single revision number. When someone talks about ˆrevision 54,˜ they're talking about a particular tree (and indirectly, the way the filesystem looked after the 54th commit).

Technically, it's not valid to talk about ˆrevision 5 of `foo.c`˜. Instead, one would say ˆ`foo.c` as it appears in revision 5.˜ Also, be careful when making assumptions about the evolution of a file. In CVS, revisions 5 and 6 of `foo.c` are always different. In Subversion, it's most likely that `foo.c` did *not* change between revisions 5 and 6.

More Disconnected Operations

In recent years, disk space has become outrageously cheap and abundant, but network bandwidth has not. Therefore, the Subversion working copy has been optimized around the scarcer resource.

The `.svn` administrative directory serves the same purpose as the `CVS` directory, except that it also stores ˆpristine˜ copies of files. This allows you to do many things off−line:

svn status — Shows you local modifications (see below)

svn diff — Shows you the details of your modifications

svn ci — Sends differences to the repository (CVS only sends fulltexts!)

svn revert — Removes your modifications

This last subcommand is new; it will not only remove local mods, but it will un−schedule operations such as adds and deletes. It's the preferred way to revert a file; running **rm file; svn up** will still work, but it blurs the purpose of updating. And, while we're on this subject

Distinction Between Status and Update

In Subversion, we've tried to erase a lot of the confusion between the **status** and **update** subcommands.

The **status** command has two purposes: (1) to show the user any local modifications in the working copy, and (2) to show the user which files are out−of−date. Unfortunately, because of CVS's hard−to−read output, many CVS users don't take advantage of this command at all. Instead, they've developed a habit of running **cvs up** to quickly see their mods. Of course, this has the side effect of merging repository changes that you may not be ready to deal with!

With Subversion, we've tried to remove this muddle by making the output of **svn status** easy to read for humans and parsers. Also, **svn update** only prints information about files that are updated, *not* local modifications.

Here's a quick guide to **svn status**. We encourage all new Subversion users to use it early and often:

svn status prints all files that have local modifications; the network is not accessed by default.

`-u` switch	Add out-of-dateness information from repository.
`-v` switch	Show *all* entries under version control.
`-n` switch	Nonrecursive.

The status command has two output formats. In the default "short" format, local modifications look like this:

```
% svn status
M     ./foo.c
M     ./bar/baz.c
```

If you specify the `--verbose` (`-v`) switch, a "long" format is used:

```
% svn status -v
M             1047    ./foo.c
_      *      1045    ./faces.html
_      *         -    ./bloo.png
M             1050    ./bar/baz.c
Head revision:   1066
```

In this case, two new columns appear. The second column contains an asterisk if the file or directory is out-of-date. The third column shows the working-copy's revision number of the item. In the example above, the asterisk indicates that `faces.html` would be patched if we updated, and that `bloo.png` is a newly added file in the repository. (The **-** next to bloo.png means that it doesn't yet exist in the working copy.)

Lastly, here's a quick summary of status codes that you may see:

```
A    Add
D    Delete
R    Replace  (delete, then re-add)
M    local Modification
U    Updated
G    merGed
C    Conflict
```

Subversion has combined the CVS **P** and **U** codes into just **U**. When a merge or conflict occurs, Subversion simply prints **G** or **C**, rather than a whole sentence about it.

Meta-data Properties

A new feature of Subversion is that you can attach arbitrary metadata to files and directories. We refer to this data as properties, and they can be thought of as collections of name/value pairs (hashtables) attached to each item in your working copy.

To set or get a property name, use the **svn propset** and **svn propget** subcommands. To list all properties on an object, use **svn proplist**.

For more information, see the section called "Properties".

Directory versions

Subversion tracks tree structures, not just file contents. It's one of the biggest reasons Subversion was written to replace CVS.

Here's what this means to you:

- The **svn add** and **svn rm** commands work on directories now, just as they work on files. So do **svn cp** and **svn mv**. However, these commands do *not* cause any kind of immediate change in the repository. Instead, the working directory is recursively ''scheduled'' for addition or deletion. No repository changes happen until you commit.
- Directories aren't dumb containers anymore; they have revision numbers like files. (Or more properly, it's correct to talk about ''directory `foo/` in revision 5''.)

Let's talk more about that last point. Directory versioning is a Hard Problem. Because we want to allow mixed–revision working copies, there are some limitations on how far we can abuse this model.

From a theoretical point of view, we define ˆrevision 5 of directory `foo`˜ to mean a specific collection of directory–entries and properties. Now suppose we start adding and removing files from `foo`, and then commit. It would be a lie to say that we still have revision 5 of `foo`. However, if we bumped `foo`'s revision number after the commit, that would be a lie too; there may be other changes to `foo` we haven't yet received, because we haven't updated yet.

Subversion deals with this problem by quietly tracking committed adds and deletes in the `.svn` area. When you eventually run **svn update**, all accounts are settled with the repository, and the directory's new revision number is set correctly. *Therefore, only after an update is it truly safe to say that you have a ˆperfect˜ revision of a directory.* Most of the time, your working copy will contain ˆimperfect˜ directory revisions.

Similarly, a problem arises if you attempt to commit property changes on a directory. Normally, the commit would bump the working directory's local revision number. But again, that would be a lie, because there may be adds or deletes that the directory doesn't yet have, because no update has happened. *Therefore, you are not allowed to commit property–changes on a directory unless the directory is up–to–date.*

Conflicts

CVS marks conflicts with in–line ˆconflict markers˜, and prints a **C** during an update. Historically, this has caused problems. Many users forget about (or don't see) the **C** after it whizzes by on their terminal. They often forget that the conflict–markers are even present, and then accidentally commit garbaged files.

Subversion solves this problem by making conflicts more tangible. See the section called ˆBasic Workcycle˜ for more details. In particular, read the section about ˆMerging others' changes˜.

Binary files

CVS users have to mark binary files with `-kb` flags, to prevent data from being munged (due to keyword expansion and line–ending translations). They sometimes forget to do this.

Subversion examines the `svn:mime-type` property to decide if a file is text or binary. If the file has no `svn:mime-type` property, Subversion assumes it is text. If the file has the `svn:mime-type` property set to anything other than `text/*`, it assumes the file is binary.

Subversion also helps users by running a binary–detection algorithm in the **svn import** and **svn add** subcommands. These subcommands will make a good guess and then (possibly) set a binary `svn:mime-type` property on the file being added. (If Subversion guesses wrong, you can always remove or hand–edit the property.)

As in CVS, binary files are not subject to keyword expansion or line–ending conversions. Also, when a binary file is

ˆmerged˜ during update, no real merge occurs. Instead, Subversion creates two files side−by−sidein your working copy; the one containing your local modifications is renamed with an `.orig` extension.

Authorization

Unlike CVS, SVN can handle anonymous and authorized users in the same repository. There is no need for an anonymous user or a separate repository. If the SVN server requests authorization when committing, the client should prompt you for your authorization (password).

Versioned Modules

Unlike CVS, a Subversion working copy is aware that it has checked out a module. That means that if somebody changes the definition of a module, then a call to **svn up** will update the working copy appropriately.

Subversion defines modules as a list of directories within a directory property. the section called ˆModules˜.

Branches and Tags

Subversion doesn't distinguish between filesystem space and ˆbranch˜ space; branches and tags are ordinary directories within the filesystem. This is probably the single biggest mental hurdle a CVS user will need to climb. Read all about it: Chapter 4

Appendix B. CVS Repository Migration

TODO Write this

Appendix C. Troubleshooting

TODO Write this

Appendix D. FAQ?

TODO Write this?

Appendix E. Other Subversion Clients

TODO Write this

Appendix F. Third Party Tools

TODO Write this

Glossary

Add
: A **svn** command that is used to add a file or directory to a repository.

Colophon

Etc.