

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

Preface	
What is Subversion	1
How This Book is Organized	1
Conventions used in this book	1
Comments and Questions	1
Acknowledgements	1
1. Introduction	
Revision Control (and what svn can do for you)	2
Target audience	2
History	2
History of Revision Control	2
History of Subversion	2
Feature list (why svn is so nice)	2
How to get svn binaries	3
How to get this book and send patches for it.	3
2. Basic Concepts	
The Repository	4
Versioning Models	5
The Problem of File-Sharing	5
The Lock-Modify-Unlock Solution	5
The Copy-Modify-Merge Solution	7
Subversion in action	8
Working copies	8
Revisions.....	10
How working copies track the repository	12
Summary	12
3. Guided Tour	
Subversion's built in help facility	14
Import	14
Initial Checkout	14
Basic Workcycle	15
Update Your Working Copy	15
Make Changes to Your Working Copy	16
Examine Your Changes	17
Resolving conflicts (Merging others' changes)	20
Commit your changes	21
Examining History	21
svn log	21
svn diff	22
Revisions.....	23
Other Commands	24
svn cleanup	24
svn info	24
svn import	24
svn export	25
svn ls	25
svn mkdir	26
4. Branching and Merging	
Branching with svn cp	27
Switching to a Branch with svn switch	28
Moving changes with svn merge	28
Rolling back a change with svn merge	29
Vendor branches	29

Removing a Branch or Tag with svn rm	32
5. Setting up a Repository	
Server Setup	34
Creating a Repository	34
Examining a Repository	35
Networking a Repository	37
Repository Maintenance	39
Repository Hooks	41
Migrating a Repository (dump/load)	43
Adding projects	43
Examples of 'svn import'	43
Vendor Branches	43
Migrating a repository	43
Partial dump/load	44
6. Advanced Topics	
Run-time Configuration Area	46
Proxies	46
Config	46
Multiple config areas	46
Properties	46
Special properties	47
Modules	50
7. Developer Information	
Layered Library Design	51
Repository Layer	52
Repository Access Layer	57
Client Layer	59
Using the APIs	59
The Apache Portable Runtime Library	60
URL and Path Requirements	60
Using Languages Other than C and C++	60
Inside the Working Copy Administration Area	62
The Entries File	62
Pristine Copies and Property Files	64
The Authentication Area	64
WebDAV	64
Programming with Memory Pools	65
Contributing to Subversion	67
Join the Community	67
Get the Source Code	67
Become Familiar with Community Policies	68
Make and Test Your Changes	68
Donate Your Changes	68
8. Complete Reference	
.....	70
A. Subversion for CVS Users	
Revision Numbers Are Different Now	71
More Disconnected Operations	71
Distinction Between Status and Update	71
Meta-data Properties	72
Directory versions	73
Conflicts	73
Binary files	73
Authorization	74
Versioned Modules	74
Branches and Tags	74
B. CVS Repository Migration	
C. Troubleshooting	

D. FAQ?
E. Other Subversion Clients
F. Third Party Tools
Glossary

List of Figures

2.1. A typical client/server system	4
2.2. The problem to avoid	5
2.3. The lock-modify-unlock solution	6
2.4. The copy-modify-merge solution	7
2.5.copy-modify-merge continued	7
2.6. The repository's filesystem	9
2.7. The repository	11
7.1. Subversion's "Big Picture"	51
7.2. Files and directories in two dimensions	54
7.3. Revisioning Time—the third dimension!	54

List of Tables

7.1. A brief inventory of the Subversion libraries 51

List of Examples

7.1. Using the Repository Layer	55
7.2. Using the Repository Layer with Python	61
7.3. A simple script to check out a working copy.	61
7.4. Contents of a typical .svn/entries file	63
7.5. Effective pool usage	66

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 revision 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-source *revision 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 revision control system as a sort of "time machine."

Some revision 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 revision control system used by the open-source community. It has a hard-earned, well-deserved reputation 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's original 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-source project 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's a 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-

regardless 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 “virtual” versioned filesystem that tracks changes to directory/file heirarchies over time. Files <i>and</i> 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

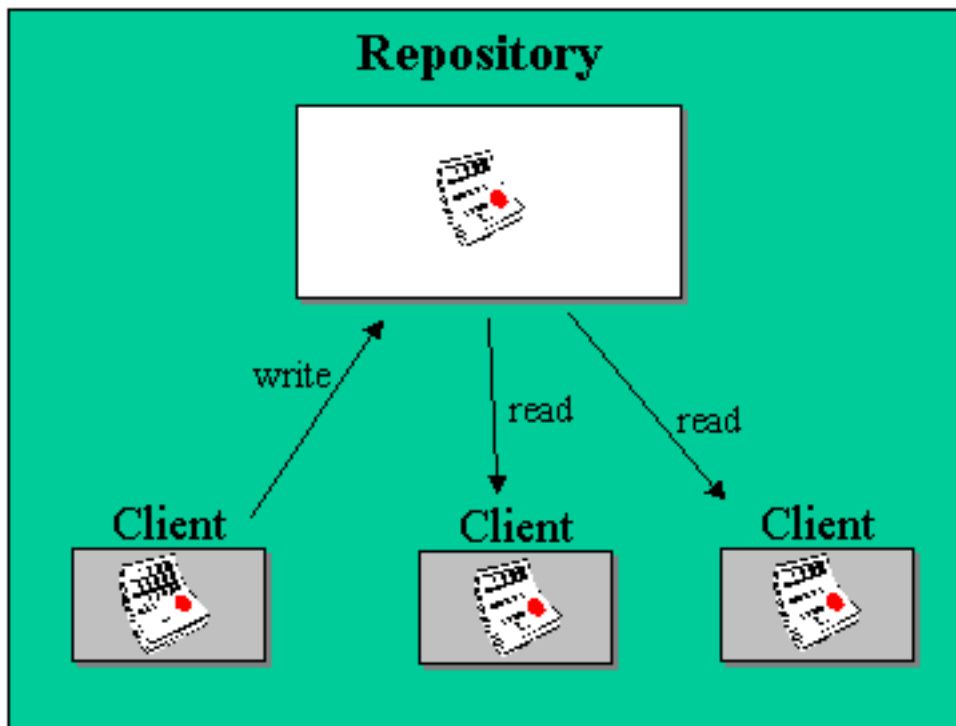
This chapter is a short, casual introduction to Subversion. If you're new to version control, this chapter is definitely for you. We begin with a discussion of general version control concepts, work our way into the specific ideas behind Subversion, and show some simple examples of Subversion in use.

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.

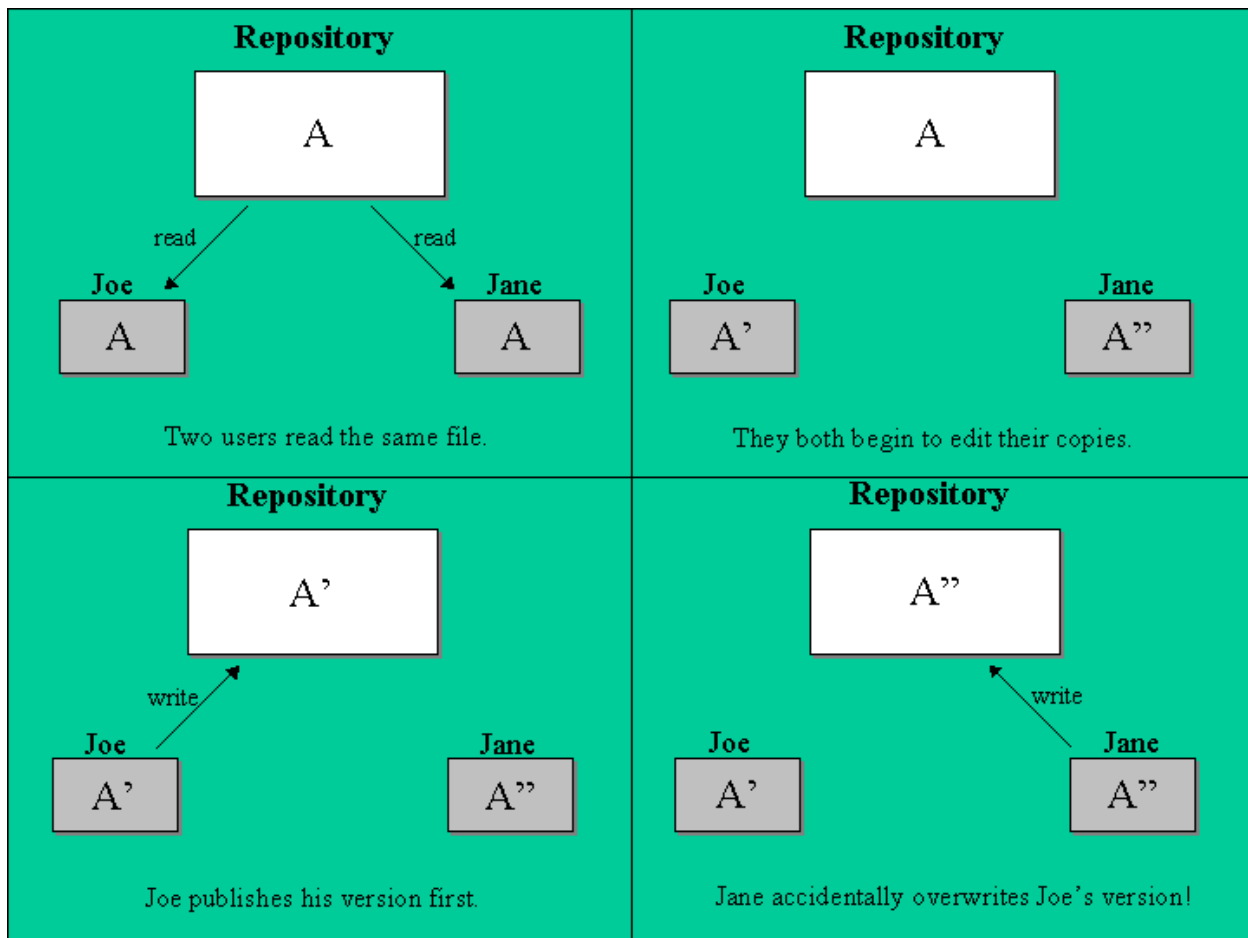
Versioning Models

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 hypothetical example. 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

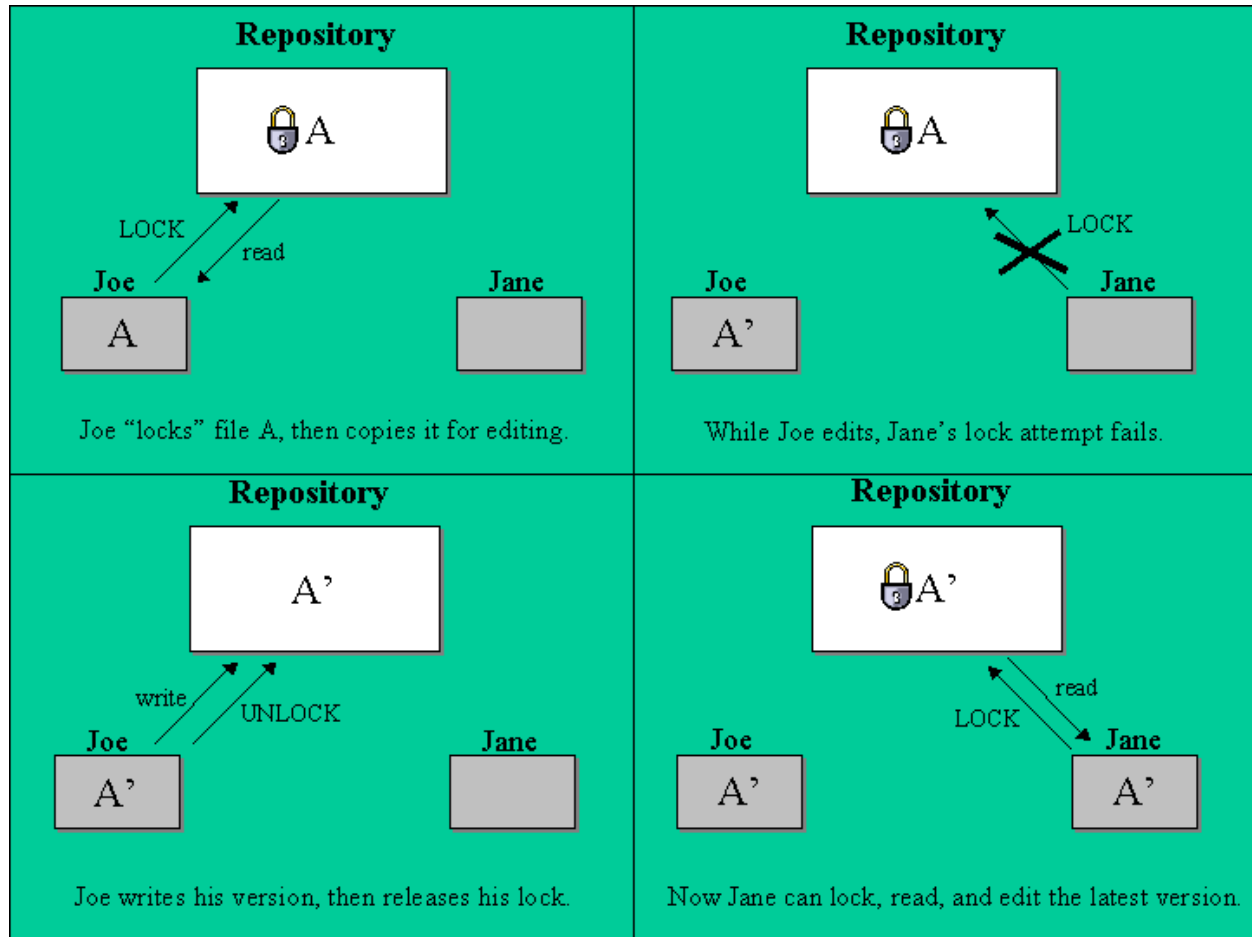


The Lock-Modify-Unlock Solution

Many version control systems use a *lock-modify-unlock* model to address this problem, which is a very simple, traditional solution. In such a system, the repository allows only one person to change a file at a time. Such a system re-

quires Joe to first "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

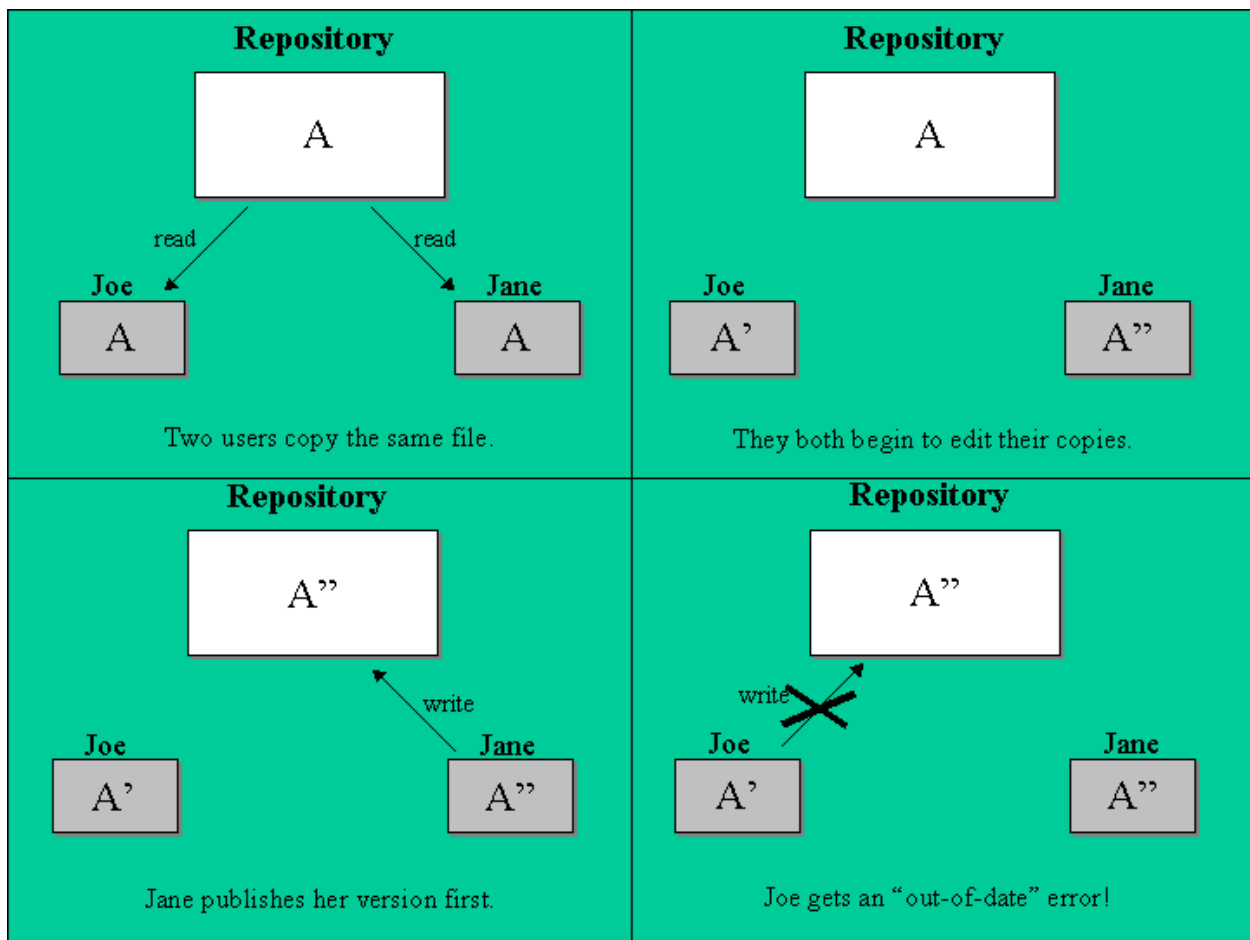
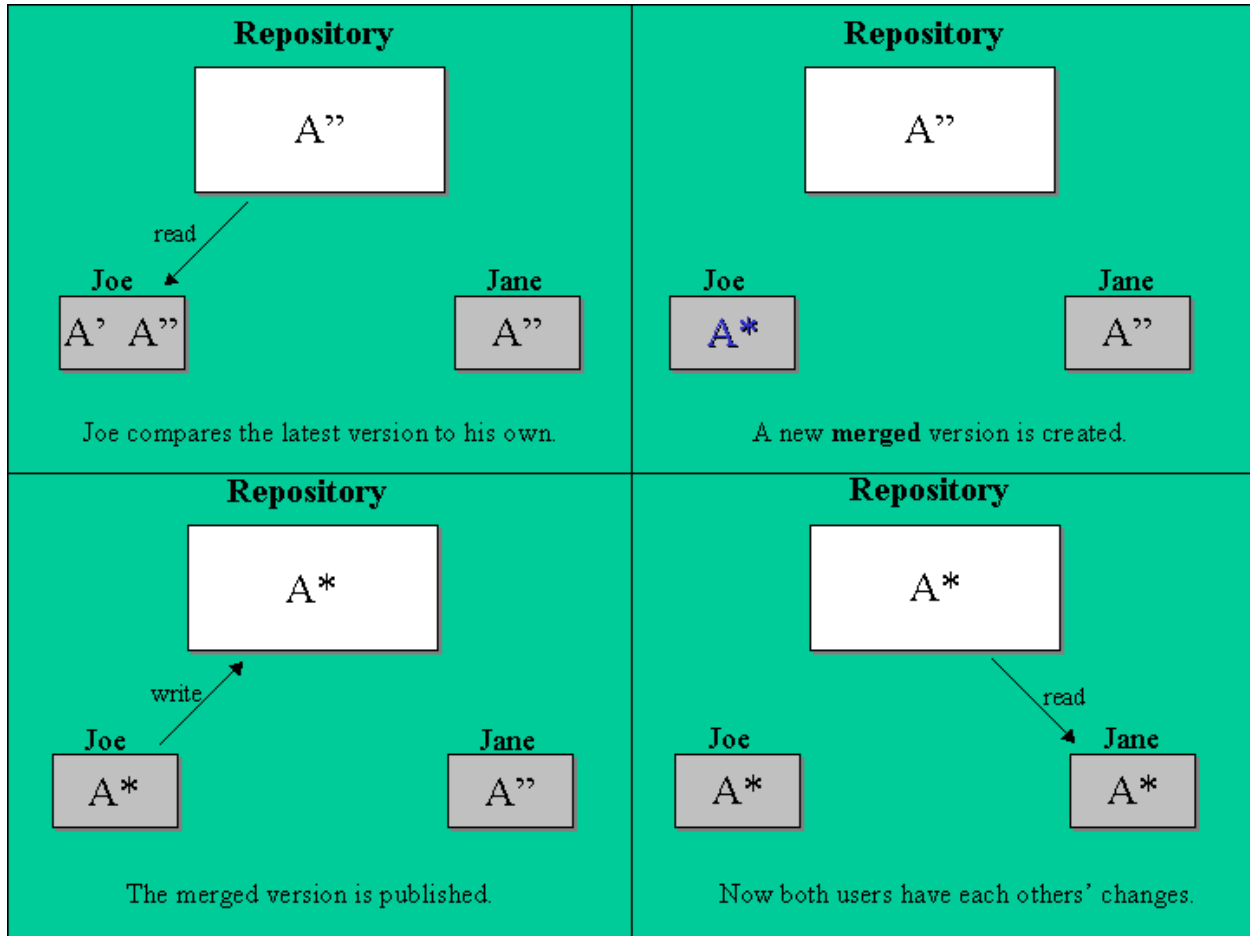


Figure 2.5. ...copy-modify-merge continued



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 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.

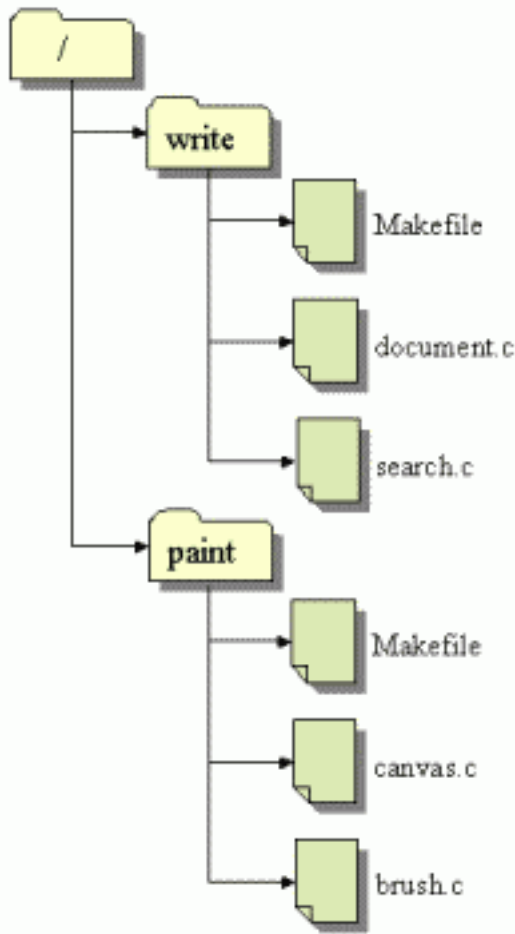
After you've made some changes to the files in your working copy and verified that they work properly, Subversion provides commands to "publish" your changes to the other people working with you on your project (by writing to the repository). If other people publish their own changes, Subversion provides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

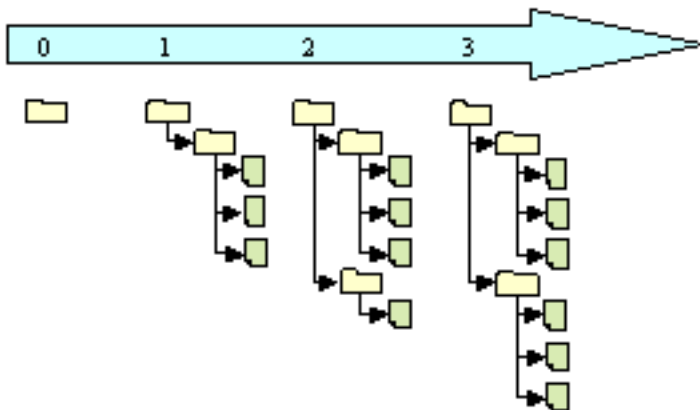
In the repository, each commit is treated as an atomic transaction: either all the commit's changes take place, or none

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 snap-"shot" 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-file revision 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/` administrative 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 revision 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 more detailed, guided tour of Subversion's commands and features.

Chapter 3. Guided Tour

How to make a lovely gumbo with your Subversion client, in 11 easy steps.

This chapter goes into more of the gritty details of client commands. For a first overview of the client's CVS-like “copy-modify-merge” model of development, see Chapter 2.

This chapter by no means covers every option to every client subcommand. Instead, it's a conversational introduction to the most common tasks you'll encounter. When in doubt, run **svn help**.

Subversion's built in help facility

Before reading on, here is the most important piece of information 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 **subcommand**.

Import

See Chapter 5.

Initial Checkout

Most of the time, you will start using a Subversion repository by doing a *checkout* of your project. “Checking out” will provide you with a local copy of the HEAD (latest revision) of the Subversion repository that you checked out.

```
$ 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.
```

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/handbook
A handbook/svn-handbook.texi
A handbook/getting_started.texi
A handbook/outline.txt
A handbook/license.texi
A handbook/repos_admin.texi
A handbook/client.texi
Checked out revision 2499.
```

Since Subversion uses a “copy-modify-merge” model instead of “lock-modify-unlock,” 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 entirely—there's no need to notify the Subversion server unless you're ready to *check in* changes, a new file, or even a directory.

Every directory in a working copy contains an *administrative area*, a subdirectory named `.svn`. Normal **ls** commands won't show this subdirectory, but it's vital. Whatever you do, don't delete or change anything in the administrative area! Subversion depends on it to manage your working copy.

You can run **svn help checkout** for command line options to checkout, although one option is very common and

worth mentioning: you can 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.
```

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 repository.

```
$ 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:

U foo	File foo was Updated (received changes from the server.)
A foo	File or directory foo was Added to your working copy.
D foo	File or directory foo was Deleted from your working copy.
R foo	File or directory foo was Replaced 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 reposi-

tory considers them to be distinct objects with distinct histories.

- G `foo` File `foo` received new changes, but also had changes of your own to begin with. The changes did not intersect, however, so Subversion has merged the repository's changes into the file without a problem.
- C `foo` File `foo` received Conflicting 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 get to work and make changes in your working copy. It's usually most convenient to create a “task” for yourself, such as writing a new feature, fixing a bug, etc. The Subversion commands that you will use here are **`svn add`**, **`svn rm`**, **`svn cp`**, and **`svn mv`**).

Changes you can make to your working copy:

- File changes This is the simplest sort of change. Unlike other revision control systems, you don't need to tell Subversion that you intend to change a file; just do it. Later on, Subversion will be able to automatically detect which files have been changed.
- Tree changes You can ask Subversion to “mark” files and directories for scheduled removal or addition. Of course, no additions or removals will happen in the repository until you decide to commit.

To make file changes, just use your normal editor, word processor, or whatever. A file needn't be in text-format; binary files work just fine.

There are at least four Subversion subcommands for making tree changes. Detailed help can be found with **`svn help`**, but here is an overview:

- `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 rm foo`** Schedule `foo` to be removed from the repository. If `foo` is a file, it immediately vanishes from the working copy—but it can be recovered with **`svn revert`** (discussed later). If `foo` is a directory, it is merely scheduled for deletion. After you commit, `foo` will no longer exist in the working copy or repository.
- `svn cp 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-history is recorded (as having originally come from `foo`.)
- `svn mv 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.

Let's amend our original statement: there *are* some use-cases that immediately commit tree changes to the repository. This usually 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. See **`svn help`** on these commands for more details.)

Examine Your Changes

So now you've finished your changes... or so you think. But what exactly did you change? How can you review them? You can use **svn status**, **svn info**, **svn diff**, and **svn revert** to find out all this and more.

Subversion has been optimized to help you with this task, and is able to do many things without talking to the repository or network at all. In particular, your working copy contains a secret cached “pristine” copy of each file within the `.svn` area. Because of this, it can quickly show you how your working files have changed, or even allow you to undo your changes.

The **svn status** command is your friend; become intimate with it. You'll probably use **svn status** more than any other command.

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:

<code>_ file_or_dir</code>	The file or directory has not been added or deleted, nor have <code>file_or_dir</code> 's contents been modified if it is a file.
<code>A file_or_dir</code>	The file or directory <code>file_or_dir</code> has been scheduled for addition into the repository.
<code>M file</code>	The contents of file <code>file</code> have been modified.
<code>D file_or_dir</code>	The file or directory <code>file_or_dir</code> has been scheduled for deletion from the repository.
<code>? file_or_dir</code>	The file or directory <code>file_or_dir</code> indicates that this file or directory is not under revision control. You can silence the question marks by either passing the <code>--quiet (-q)</code> switch to svn status , or by setting the <code>svn:ignore</code> property on the parent directory, see Chapter 6.
<code>! file_or_dir</code>	The file or directory <code>file_or_dir</code> is under revision control but the working copy is missing. 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.
<code>~ file_or_dir</code>	The file or directory <code>file_or_dir</code> is under revision control as one kind of object, but what's actually on disk is some other kind. For example, Subversion might be expecting a file, but the user has removed the file and created a directory in its place, without using the svn rm nor svn add commands.

The second column tells the status of a file's or directory's properties, see Chapter 6. If a `M` appears in the second column, then the properties have been modified, otherwise a whitespace will be printed. If only the properties of a file or directory are modified, then you will get `_M` printed in the first and second columns. The first `_` is just printed to make it clear to the eye that the properties are modified and not the contents.

The third column will only show whitespace or a `L` which means that `svn` has locked the item in the `.svn` working area. You will see `L` if you run `svn status` in a directory you are currently running `svn commit` when you are editing the log message. If there are no running `@command{svn}`'s, then presumably `@command{svn}` was forcibly quit or died and the lock needs to be cleaned up by running `svn cleanup`. 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 mv` or `svn cp` 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. `_ @ @ +` means the item is part of a subtree scheduled for addition-with-history, i.e. some parent got copied, and its just coming along for the ride. `M @ @ +` means the item is part of a subtree scheduled for addition-with-history, *and* it has local mods. When you commit, first some parent will be added-with-history (copied), which means this file will automatically exist in the copy. Then the local mods will be further uploaded into the copy.

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 irrespective of `svn status` and `svn:ignore`'s regular expressions, then use the `--no-ignore` command line option.

If a single path is passed to the command, it will tell you about it:

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

This command also has a `--verbose (-v)` mode, which will show you the status of *every* item in your working copy:

```
$ svn status -v
M      44      23      joe      ./README
_      44      30      frank   ./INSTALL
M      44      20      frank   ./bar.c
_      44      18      joe     ./stuff
_      44      35      mary    ./stuff/trout.c
D      44      19      frank   ./stuff/fish.c
_      44      21      mary    ./stuff/things
A      0        ?        ?       ./stuff/things/bloo.h
_      44      36      joe     ./stuff/things/gloo.c
```

This is the “long form” output of `svn status`. The first column is still the same. The second column shows the working-revision of the item. The third and fourth column show the revision in which the item last changed, and who changed it.

All of the above invocations to `svn status` do not contact the repository, they work only locally by comparing the metadata in the `.svn` directory with the working area.

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

```
$ svn status -u -v
M      *      44      23      joe      ./README
M      *      44      20      frank   ./bar.c
_      *      44      35      mary    ./stuff/trout.c
```

```

D          44          19    frank    ./stuff/fish.c
A          0           ?     ?       ./stuff/things/bloo.h

```

Notice the two asterisks: if you were to run **svn up** at this point, you would receive changes to `README` and `trout.c`. Hmm, better be careful. You'll need to absorb those server-changes on `README` before you commit, lest the repository reject your commit for being out-of-date. (More on this subject later).

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 “pristine” 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.

Now suppose you see this output, and realize that your changes to `README` are a mistake; perhaps you accidentally typed that text into the wrong file in your editor.

The `svn revert` command is exactly for this purpose. It throws away all changes to your file:

```

$ svn revert README
Reverted ./README

```

The file is reverted to its pre-modified state by overwriting it with the cached “pristine” copy. But also note that **svn revert** can undo any scheduled operations—in case you decide that you don't want to add a new file after all, or that

you don't want to remove something.

All three of these commands (**svn status**, **svn diff**, **svn revert**) can be used without any network access (except for the `-u` switch to status). This makes it easy to manage your changes-in-progress while traveling on an airplane, etc.

Subversion manages this by keeping private caches of pristine versions of each versioned file inside of the `.svn` administrative 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 that *entire file* to the server in order to relay the local modifications.

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 up
U  ./INSTALL
G  ./README
C  ./bar.c
```

The `U` and `G` codes are nothing to sweat about; those files cleanly absorbed changes from the repository. The `G` stands for merged, which means that the file had local changes to begin with, but the repository changes didn't overlap in any way.

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

Whenever a conflict occurs:

- A `C` is printed during the update, and Subversion remembers that the file is "conflicted."
- Conflict markers are placed into the file, to visibly demonstrate the overlapping areas.
- Three fulltext files are created; these files are the original three files that could not be merged together. Given a file named `ORIG_NAME`, the three new fulltext files have filenames of the form `ORIG_NAME.*.mine`, `ORIG_NAME.*.rOLD_REV` and `ORIG_NAME.*.rNEW_REV`. Here `*` represents some random digits that SVN chooses, `ORIG_NAME.*.mine` is a copy of the file that existed in your local working copy before the merge and without any conflict markers, `ORIG_NAME.*.rOLD_REV` is the original version of `ORIG_NAME` at the revision that your working copy is based off of with `OLD_REV` replaced with the specific revision number, and `ORIG_NAME.*.rNEW_REV` is the original version of `ORIG_NAME` that the file is being merged to, again with `NEW_REV` replaced with the specific revision number.

At this point, Subversion will *not* allow you to commit the file until the three temporary files are removed.

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** to toss all of your changes.

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

Commit your changes

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

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 permanently attached to the new revision you create.

```
$ svn commit -m "Added include lines and corrected # of cheese slices."
Sending          bar.c
Transmitting file data .
Committed revision 3.
$
```

Another way to specify a log message is to place it in a file, and pass the filename with the `-F` switch. If you fail to specify either the `@option{-m}` or `-F` switch, then Subversion will automatically launch your favorite **\$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 nobody 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 that one or more of your files is out-of-date. At this point, you need to run **svn update** again, deal with any merges or conflicts that result, and attempt your commit again.

That covers the most basic work cycle for using Subversion. Run **svn help <commandname>** for help on any of the commands covered in this section.

Examining History

As we mentioned earlier, the repository is like a time machine. It remembers every revision ever committed, and allows you to explore this history.

There are two commands that mine 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.

svn log

To find out information about the history of a file or directory, you use the **svn log** command. **svn log** will tell you who made changes to a file and at what revision, the time and date of that revision, and the log message that accompanied the commit.

```
$ svn log
-----
rev 3:  fitz | Mon, 15 Jul 2002 18:03:46 -0500 | 1 line
Added include lines and corrected # of cheese slices.
-----
rev 2:  someguy | Mon, 15 Jul 2002 17:47:57 -0500 | 1 line
Added main() methods.
-----
rev 1:  fitz | 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 -r 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 on a single file or directory. The commands

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

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

And while we're on the subject, **svn log** also takes a `--verbose (-v)` switch too; it includes a list of changed-paths in each revision:

```
$ svn log -r 8 -v
-----
rev 8:  jrandom | 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** in an previous section; it displays file differences in unified diff format. Earlier, it was used to show the local modifications made to our working copy.

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

Examining local changes

Invoking **svn diff** with no switches will compare your working files to the cached “pristine” copies in the `.svn` area:

```
$ svn diff foo
Index: ./foo
=====
--- ./foo
+++ ./foo Tue Jul 16 15:19:53 2002
@@ -1,2 @@
An early version of the file
+...extra edits
```

Comparing working copy to repository

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

```
$ svn diff -r 3 foo
```

```
Index: ./foo
=====
--- ./foo
+++ ./foo Tue Jul 16 15:19:53 2002
@@ -1,2 +1,2 @@
An early version of the file
-Second version of the file
+...extra edits
```

Comparing repository to repository

If two revision numbers are passed via `-r`, then the two revisions are directly compared.

```
$ svn diff -r 2:3 foo

Index: ./foo
=====
--- ./foo
+++ tmp.280.00001 Tue Jul 16 15:22:19 2002
@@ -1 +1,2 @@
An early version of the file
+Second version of the file
```

If you read the help for `svn diff`, you'll discover that you can supply URLs instead of working copy paths as well. This is especially useful if you wish to inspect changes when you have no working copy available:

```
$ svn diff -r 23:24 http://foo.com/some/project
...
```

Revisions

As you may have noticed, many Subversion commands are able to process the `-r` switch. Here we describe 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 `-r` switch, and are resolved into specific revision numbers:

HEAD The latest revision in the repository.

BASE The “pristine” 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 -r PREV:COMMITTED foo.c
# shows the last change committed to foo.c

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

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

$ svn diff -r BASE:HEAD foo.c
```

```
# compares your "pristine" foo.c (no local mods) with the latest version
# in the repository

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

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

Other Commands

svn cleanup

When Subversion modifies your working copy (or any information within `.svn`), it tries to do so as safely as possible. Before changing anything, it writes its intentions to a logfile, then executes the commands in the logfile. It's similar in design to a journaled filesystem; if the user hits Control-C or if the machine crashes, the logfiles are left lying around. By re-executing the logfiles, the work can complete, 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 re-runs any leftover logs, removing locks in the process. Use this command if Subversion ever tells you that some part of your working copy is “locked”. Also, **svn status** will display an `L` next to locked items:

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

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

svn info

In general, we try to discourage users from directly reading the `.svn/entries` file used to track items. Instead, curiosity can be quelled by using the **svn info** to display most of the tracked information:

```
$ svn info client.texi
Path: client.texi
Name: client.texi
Url: http://svn.collab.net/repos/svn/trunk/doc/handbook/client.texi
Revision: 2548
Node Kind: file
Schedule: normal
Last Changed Author: fitz
Last Changed Rev: 2545
Last Changed Date: 2002-07-15 23:03:54 -0500 (Mon, 15 Jul 2002)
Text Last Updated: 2002-07-16 08:48:04 -0500 (Tue, 16 Jul 2002)
Properties Last Updated: 2002-07-16 08:48:03 -0500 (Tue, 16 Jul 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

```
/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.

```
$ 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 ls

The `ls` command lets you find what files are in a repository directory.

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

If you want a more detailed listing, pass the `-v` flag and you will get output like this.

```
$ svn ls -v 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 (“P” if it does, “_” if it doesn't), the revision it was last updated at, the user who last updated it, it's size, the date it was last updated, and the filename.

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 -m "made new dir"
Committed revision 1123.
```

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

Chapter 4. Branching and Merging

Branches and tags are general concepts common to almost all revision control systems. If you're not familiar with these ideas, you can find a good introductory explanation in Karl Fogel's free CVS book: http://cvsbook.red-bean.com/cvsbook.html#Branching_Basics

At this point, you should understand how each commit creates an entire new filesystem tree in the repository. (If not, read revisions, Chapter 2, Chapter 2.)

As you may have suspected, the filesystem doesn't grow 652 new inodes each time a new revision is created. Instead, each new tree is *mostly* made of pointers to already-existing nodes; new nodes are created only for changed items, and all the rest of the revision tree is “shared storage” with other revision trees. This technique demonstrates how the filesystem is able to make “cheap copies” of things. These cheap copies are nothing more than directory entries that point to existing nodes. And this is the basis of tags and branches.

Branching with `svn cp`

Suppose we have a repository whose head tree is revision 82. In this repository is a subdirectory `mooIRC` that contains a software project that is ready to be tagged. How do we tag it? Very simple: make a “cheap” copy of this directory. In other words, create a new directory entry (somewhere else in the filesystem) that points to this *specific* node that represents directory `mooIRC` in revision 82. Of course, you can name the new directory entry whatever you want -- probably a tag-name like `mooIRC-beta`.

The easiest way to make this copy is with `svn cp`, which, incidentally, can operate entirely on URLs, so that the copy happens only on the server-side:

```
$ svn cp http://foo.com/repos/mooIRC http://foo.com/repos/mooIRC-beta
Committed revision 83.
```

Now, as long as you never touch the contents of the directory `mooIRC-beta`, that entry will forever point to a node that looks the way `mooIRC` did at a specific moment in time (however it looked in revision 82). And that's exactly what a *tag* is.

But suppose `mooIRC-beta` isn't sacred, and instead you decide to start making commits to it. And suppose you *also* continue to make commits in the original `mooIRC` directory. Then you have two directories that started out looking identical -- their common ancestor was `mooIRC` in revision 82 -- but now have diverged their contents over time. In other words, they represent different *branches* of the project.

It's very important to note that the Subversion filesystem is *not* aware of “tags” or “branches.” It's only aware of directories, and all directories are equal. The tag and branch concepts are purely *human* meanings attached to particular directories.

For this reason, it's up to users (and the Subversion repository administrator) to choose sane policies that help elucidate these labels. For example, here's a good way to lay out your repository:

```
/
/projectA
/projectA/trunk/
/projectA/branches/
/projectA/tags/
/projectB
/projectB/trunk/
/projectB/branches/
/projectB/tags/
```

Each time `/projectA/trunk` reaches a taggable state, make a copy of the directory somewhere in `/`

projectA/tags/, and set the copy to read-only. Use the same procedure to create a branch in /projectA/branches/.

An alternate way to lay out a repository:

```
/
/trunk
/trunk/projectA
/trunk/projectB
/branches
/branches/projectA
/branches/projectB
/tags
/tags/projectA
/tags/projectB
```

Or, of course, you could just place each project into a dedicated repository. It's up to you. For examples on how to create a repository with one of these structures, Chapter 5.

Switching to a Branch with `svn switch`

The **svn switch** command allows you to “move” some or all of your working copy to a branch or tag. For example, suppose I have a working copy of mooIRC, and I'd like to work on some subsystem as it appears in a subdirectory of mooIRC-beta. At the same time, I want the rest my working copy to remain on the original mooIRC branch. To do this, I switch the appropriate subdir to the new branch location:

```
$ svn switch http://foo.com/repos/mooIRC-beta/subsystems/renderer \
    mooIRC/subsystems/renderer

U mooIRC/subsystems/renderer/foo.c
U mooIRC/subsystems/renderer/bar.h
U mooIRC/subsystems/renderer/baz.c
```

Now my working copy of the `renderer` subdirectory represents a different location on the server.

Really, **svn switch** is just a fancier version of **svn update**. Whereas **svn update** has the ability to move your working copy through time (either by updating to the latest revision, or by updating to a specific revision given with `-r`), **svn switch** is able to move your working copy through time *and* space.

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.

Moving changes with `svn merge`

Suppose a team of programmers working on the mooIRC-beta branch have fixed a critical bug, and the team working on the original mooIRC branch would like to apply that change as well.

The **svn merge** command is the answer. You can think of **svn merge** as a special kind of **svn diff**; only instead of displaying unified diffs to the screen, it *applies* the differences to your working copy as if they were local changes.

For example, suppose the bug fix happened in a commit to the mooIRC-beta branch in revision 102.

```
$ svn diff -r 101:102 http://foo.com/repos/mooIRC-beta
... # diffs sent to screen

$ svn merge -r 101:102 http://foo.com/repos/mooIRC-beta mooIRC
U mooIRC/glorb.c
```

```
U mooIRC/src/floo.h
```

While the output of **svn merge** looks similar to **svn update** or **svn switch**, it is in fact only applying temporary changes to the working files. Once the differences are applied as local changes, you can examine them as usual with **svn diff**, **svn status**, or undo them with **svn revert** as usual. If the changes are acceptable, you can commit them.

Rolling back a change with svn merge

Another common use for **svn merge** is for rolling back a change that has been committed. Say you commit some changes in revision 10, and later decide that they were a mistake. You can easily revert the tree to the state it was in at revision 9 with an **svn merge** command.

```
$ svn commit -m "change some stuff"
Sending          bar.c
Sending          foo.c
Transmitting file data ..
Committed revision 10.
$

... # developer continues on and realizes he made a mistake

$ svn merge -r 10:9 .
U ./bar.c
U ./foo.c
$ svn commit -m "oops, reverting revision 10"
Sending          bar.c
Sending          foo.c
Transmitting file data ..
Committed revision 11.
```

If you aren't rolling back the changes to your current directory (say you want to roll back one specific file, or all the files in one specific subdirectory), then the syntax is slightly different, as you have to tell **svn merge** where it should merge the changes into.

```
$ svn merge -r 10:9 baz/ baz/
U ./baz/bar.c
U ./baz/foo.c
$ svn commit -m "reverting revision 10's changes in baz/"
Sending          baz/bar.c
Sending          baz/foo.c
Transmitting file data ..
Committed revision 12.
$

... # developer continues on and later makes another mistake

$ svn merge -r 13:12 baz/foo.c baz/foo.c
U ./baz/foo.c
$ svn commit -m "reverting revision 12's change to foo.c"
Sending          baz/foo.c
Transmitting file data .
Committed revision 15.
```

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.

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 “vendor branch”. Subversion doesn't have a for-

mal “vendor 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-1.0 release 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-1.1 would 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-1.1` in `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 `@emph{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 `@emph{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`.

Removing a Branch or Tag with `svn rm`

The `svn rm` command can operate on URLs. A file or directory can be “remotely” deleted from the repository, with no working copy present:

```
$ svn rm http://foo.com/repos/tags/mooIRC-bad-tag -m "deleting bad tag"
```

Committed revision 1023.

Of course, this is still a form of immediate commit, so some kind of log message is still required.

Enough said!

Chapter 5. Setting up a Repository

How to administer a Subversion repository.

In this chapter, we'll mainly focus on how to use the **svnadmin** and **svnlook** programs to work with repositories.

Server Setup

Creating a Repository

Creating a repository is incredibly simple:

```
$ svnadmin create path/to/myrepos
```

This creates a new repository in a subdirectory `myrepos`.

(Note that the **svnadmin** and **svnlook** programs operate *directly* on a repository, by linking to `libsvn_fs.so`. So these tools expect ordinary, local paths to the repositories. This is in contrast with the **svn** client program, which always accesses a repository via some URL, whether it be via `http://` or `file:///` schemas.)

A new repository always begins life at revision 0, which is defined to be nothing but the root (`/`) directory.

As mentioned earlier, repository revisions can have unversioned properties attached to them. In particular, every revision is created with a `svn:date` timestamp property. (Other common properties include `svn:author` and `svn:log`)

For a newly created repository, revision 0 has nothing but a `svn:date` property attached.

Here is a quick run-down of the anatomy of a repository:

```
$ ls myrepos
conf/
dav/
db/
hooks/
locks/
```

- `conf` Currently unused; repository-side config files will go in here someday.
- `dav` If the repository is being accessed by Apache and `mod_dav_svn`, some private housekeeping databases are stored here.
- `db` The main Berkeley DB environment, full of DB tables that comprise the data store for `libsvn_fs`. This is where all of your data is! In particular, most of your files' contents end up in the "strings" table. Logfiles accumulate here as well, so transactions can be recovered.
- `hooks` Where pre-commit and post-commit hook scripts live. (And someday, read-hooks.)
- `locks` A single file lives here; repository readers and writers take out shared locks on this file. Do not remove this file.

Once the repository has been created, it's very likely that you'll want to use the `svn` client to import an initial tree. See Chapter 5.)

You may want to give your repository an initial directory structure that reflects the trunk, branches, and tags of your project(s) (See Chapter 4.) You can do this via **svn mkdir**:

```
$ svnadmin create /path/to/repos
$ svn mkdir file:///path/to/repos/projectA -m 'Base dir for A'
Committed revision 1.

$ svn mkdir file:///path/to/repos/projectA/trunk -m 'Main dir for A'
Committed revision 2.

$ svn mkdir file:///path/to/repos/projectA/branches -m 'Branches for A'
Committed revision 3.

$ svn mkdir file:///path/to/repos/projectA/tags -m 'Tags for A'
Committed revision 4.

$ svn co file:///path/to/repos/projectA/trunk projectA
Checked out revision 4.

# ... now work on projectA ...
```

With **svn import**, you can create the structure with a single commit:

```
$ svnadmin create /path/to/repos
$ mkdir projectA
$ mkdir projectA/trunk
$ mkdir projectA/branches
$ mkdir projectA/tags
$ svn import file:///path/to/repos projectA projectA -m 'Dir layout for A'
Adding      projectA/trunk
Adding      projectA/branches
Adding      projectA/tags
Committed revision 1.

$ rm -rf projectA/
$ svn co file:///path/to/repos/projectA/trunk projectA
Checked out revision 1.

# ... now work on projectA ...
```

Examining a Repository

Transactions and Revisions

A Subversion repository is essentially a sequence of trees; each tree is called a **revision**. (If this is news to you, it might be good for you to see Chapter 2.)

Every revision begins life as a **transaction tree**. When doing a commit, a client builds a transaction that mirrors their local changes, and when the commit succeeds, the transaction is effectively “promoted” into a new revision tree, and is assigned a new revision number.

At the moment, updates work in a similar way: the client builds a transaction tree that is a “mirror” of their working copy. The repository then compares the transaction tree with some revision tree, and sends back a tree-delta. After the update completes, the transaction is deleted.

Transaction trees are the only way to “write” to the repository's versioned filesystem; all users of `libsvn_fs` will do this. 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 aborted if the commit fails.) In the case of an error or bug, it's possible that a transaction can be accidentally left lying around -- the `libsvn_fs` caller might die before deleting it. And in theory, someday whole workflow applications might revolve around the creation of transactions; they might be examined in

turn by different managers before being deleted or promoted to revisions.

The point is: if you're administering a Subversion repository, you're going to have to examine revisions and transactions. It's part of monitoring the health of the repository.

svnlook

svnlook is a read-only tool that can be used to examine the revision and transaction trees within a repository. Its useful for system administrators, and can be used by the `pre-commit` and `post-commit` hook scripts as well.

The simplest usage is

```
$ svnlook repos
```

This will print information about the `HEAD` revision in the repository “`repos`”. In particular, it will show the log message, author, date, and a diagram of the tree.

To look at a particular revision or transaction:

```
$ svnlook repos rev 522
$ svnlook repos txn 340
```

Or, if you only want to see certain types of information, **svnlook** accepts a number of subcommands. For example,

```
$ svnlook repos rev 522 log
$ svnlook repos rev 559 diff
```

Available subcommands are:

`log` Print the tree's log message.

`author` Print the tree's author.

`date` Print the tree's datestamp.

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

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

`diff` Print unified diffs of changed files.

The Shell

The **svnadmin** tool has a toy “shell” mode as well. It doesn't do much, but it allows you to poke around the repository as if it were an imaginary mounted filesystem. The basic commands `cd`, `ls`, `exit`, and `help` are available, as well as the very special command `cr`—“change revision”. The last command allows you to move *between* revision trees.

```
$ svnadmin shell repos
<609: />$
<609: />$ ls
< 1.0.2i7> [ 601] 1          0  trunk/
```

Why read-only? Because if a pre-commit hook script changed the transaction before commit, the working copy would have no way of knowing what happened, and would therefore be out of sync and not know it. Subversion currently has no way to handle this situation, and maybe never will.

```

<nh.0.2i9> [ 588] 0          0  branches/
<jz.0.18c> [ 596] 0          0  tags/

<609: />$ cd trunk
<609: /trunk>$ cr 500
<500: /trunk>$ ls
< 2.0.1> [ 1] 0          3462  svn_config.dsp
< 4.0.dj> [ 487] 0          3856  PORTING
< 3.0.cr> [ 459] 0          7886  Makefile.in
< d.0.ds> [ 496] 0          9736  build.conf
< 5.0.d9> [ 477] 1          0      ac-helpers/
< y.0.1> [ 1] 0          1805  subversion.dsp
...
<500: />$ exit

```

The output of `ls` has only a few columns:

NODE-ID	CREATED-REV	HAS_PROPS?	SIZE	NAME
< 1.0.2i7>	[601]	1	0	trunk/
<nh.0.2i9>	[588]	0	0	branches/
<jz.0.18c>	[596]	0	0	tags/

Networking a Repository

Okay, so now you have a repository, and you want to make it available over a network.

Subversion's primary network server is Apache `httpd` speaking WebDAV/deltaV protocol, which is a set of extension methods to `http`. (For more information on DAV, see <http://www.webdav.org/>.)

To network your repository, you'll need to

- Get Apache `httpd` 2.0 up and running with the `mod_dav` module.
- Install the `mod_dav_svn` plugin to `mod_dav`, which uses Subversion's libraries to access the repository.
- Configure your `httpd.conf` file to export the repository.

You can accomplish the first two items by either building `httpd` and Subversion from source code, or by installing a binary packages on your system. The second appendix of this document contains more detailed instructions on doing this. (Chapter 5.) Instructions are also available in the `INSTALL` file in Subversion's source tree.

In this section, we focus on configuring your `httpd.conf`.

Somewhere near the bottom of your configuration file, define a new `<Location>` block:

```

<Location /repos/myrepo>
  DAV svn
  SVNPath /absolute/path/to/myrepo
</Location>

```

This now makes your `myrepo` repository available at the URL `http://hostname/repos/myrepo`.

Alternately, you can use the `SVNParentPath` directive to indicate a “parent” directory whose immediate subdirectories are assumed to be independent repositories:

```

<Location /repos>
  DAV svn

```

```
SVNParentPath /absolute/path/to/parent/dir
</Location>
```

If you were to run **svnadmin create foorepo** within this parent directory, then the url `http://hostname/repos/foorepo` would automatically be accessible without having to change `httpd.conf` or restart `httpd`.

Note that this simple `<Location>` setup starts life with no access restrictions at all:

- Anyone can use their `svn` client to checkout either a working copy of a repository URL, or of any URL that corresponds to a subdirectory of a repository.
- By pointing an ordinary web browser at a repository URL, anyone can interactively browse the repository's latest revision.
- Anyone can commit to a 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.

First, create an empty file that will hold `httpd` usernames and passwords. Place names and crypted passwords into this file like so:

```
joe:Msr3lKOsYMkpc
frank:Ety6rZX6P.Cqo
mary:kV4/mQbu0iq82
```

You can generate the crypted passwords by using the standard `crypt(3)` command, or using the **htpasswd** tool supplied in Apache's `bin` directory:

```
$ /usr/local/apache2/bin/htpasswd -n sussman
New password:
Re-type new password:
sussman:kUqncD/TBbdC6
```

Next, add lines within your `<Location>` block that point to the user file:

```
AuthType Basic
AuthName "Subversion repository"
AuthUserFile /path/to/users/file
```

If you want to restrict *all* access to the repository, add one more line:

```
Require valid-user
```

This line makes Apache require user authentication for every single type of `http` request to your repository.

To restrict write-access only, you need to require a valid user for all request methods *except* those that are read-only:

```
<LimitExcept GET PROPFIND OPTIONS REPORT>
  Require valid-user
</LimitExcept>
```

Or, if you want to get fancy, you can create two separate user files, one for readers, and one for writers:

```
AuthGroupFile /my/svn/group/file

<LimitExcept GET PROPFIND OPTIONS REPORT>
  Require group svn_committers
</LimitExcept>

<Limit GET PROPFIND OPTIONS REPORT>
  Require group svn_committers
  Require group svn_readers
</Limit>
```

These are only a few simple examples. For a complete tutorial on Apache access control, please consider taking a look at the “Security” tutorials found at <http://httpd.apache.org/docs-2.0/misc/tutorials.html>.

Another note: in order for **svn cp** to work (which is actually implemented as a DAV COPY request), `mod_dav` needs to be able to determine the hostname of the server. A standard way of doing this is to use Apache's `ServerName` directive to set the server's hostname. Edit your `httpd.conf` to include:

```
ServerName svn.myserver.org
```

If you are using virtual hosting through Apache's `NameVirtualHost` directive, you may need to use the `ServerAlias` directive to specify additional names that your server is known by.

(If you are unfamiliar with an Apache directive, or not exactly sure about what it does, don't hesitate to look it up in the documentation: <http://httpd.apache.org/docs-2.0/mod/directives.html>.)

You can test your exported repository by firing up `httpd`:

```
$ /usr/local/apache2/bin/apachectl stop
$ /usr/local/apache2/bin/apachectl start
```

Check `/usr/local/apache2/logs/error_log` to make sure it started up okay. Try doing a network checkout from the repository:

```
$ svn co http://localhost/repos wc
```

The most common reason this might fail is permission problems reading the repository db files. Make sure that the user `nobody` (or whatever `UID` the `httpd` process runs as) has permission to read and write the Berkeley DB files! This is a very common problem.

You can see all of `mod_dav_svn`'s complaints in the Apache error logfile, `/usr/local/apache2/logs/error_log`, or wherever you installed Apache. For more information about tracing problems, see "Debugging the server" in the `HACKING` file.

Repository Maintenance

Berkeley DB Management

At the time of writing, 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 `repos/db/`.

Berkeley DB comes with a number of tools for managing these files, and they have their own excellent documentation. (See <http://www.sleepycat.com/>, or just read the BDB man pages.) We won't cover all of these tools

here; rather, we'll mention just a few of the more common procedures that repository administrators might need.

First, remember that Berkeley DB has genuine transactions. Every attempt to change the DB is first logged. If anything ever goes wrong, the DB can back itself up to a previous `checkpoint` and replay transactions to get the data back into a sane state.

In our experience, we have seen situations where a bug in Subversion (which causes a crash) can sometimes have a side-effect of leaving the DB environment in a “locked” state. Any further attempts to read or write to the repository just sit there, waiting on the lock.

To “unwedge” the repository:

1. Shut down the Subversion server, to make sure nobody is accessing the repository's Berkeley DB files.
2. Switch to the user who owns and manages the database.
3. Run the command `db_recover -v -h /path/to/repos/db`, where `repos` is the repository's directory name. You should see output like this:

```
db_recover: Finding last valid log LSN: file: 40 offset 4080873
db_recover: Checkpoint at: [40][4080333]
db_recover: Checkpoint LSN: [40][4080333]
db_recover: Previous checkpoint: [40][4079793]
db_recover: Checkpoint at: [40][4079793]
db_recover: Checkpoint LSN: [40][4079793]
db_recover: Previous checkpoint: [40][4078761]
db_recover: Recovery complete at Sun Jul 14 07:15:42 2002
db_recover: Maximum transaction id 80000000 Recovery checkpoint [40][4080333]
```

Make sure that the `db_recover` program you invoke is the one distributed with the same version of Berkeley DB you're using in your Subversion server.

4. Restart the Subversion server.

Make sure you run this command as the user that owns and manages the database—typically your Apache process—and *not* as root. Running `db_recover` as root leaves files owned by root in the `db` directory, which the non-root user that manages the database cannot open. If you do this, you'll get “permission denied” error messages when you try to access the repository.

Second, a repository administrator may need to manage the growth of logfiles. At any given time, the DB environment is using at least one logfile to log transactions; when the “current” logfile grows to 10 megabytes, a new logfile is started, and the old one continues to exist.

Thus, after a while, you may see a whole group of 1MB logfiles lying around the environment. At this point, you can make a choice: if you leave every single logfile behind, it's guaranteed that `db_recover` will always be able to replay every single DB transaction, all the way back to the first commit. (This is the “safe”, or perhaps paranoid, route.) On the other hand, you can ask Berkeley DB to tell you which logfiles are no longer being actively written to:

```
$ db_archive -a -h repos/db
log.0000000023
log.0000000024
log.0000000029
```

Subversion's own repository uses a `post-commit` hook script, which, after performing a “hot-backup” of the repository, removes these excess logfiles. (In the Subversion source tree, see `tools/backup/hot-backup.py`).

This script also illustrates the safe way to perform a backup of the repository while it's still up and running: recursively copy the entire repository directory, then re-copy the logfiles listed by **db_archive -l**.

To start using a repository backup that you've restored, be sure to run **db_recover -v** command in the db area first. This guarantees that any unfinished log transactions are fully played before the repository goes live again. (The **hot-backup.py** script does that for you during backup, so you can skip this step if you decide to use it.)

Finally, note that Berkeley DB has a whole locking subsystem; in extremely intensive svn operations, we have seen situations where the DB environment runs out of locks. The maximum number of locks can be adjusted by changing the values in the `repos/db/DB_CONFIG` file. Don't change the default values unless you know what you're doing; be sure to read <http://www.sleepycat.com/docs/ref/lock/max.html> first.

Tweaking with Svnadmin

The **svnadmin** tool has some subcommands that are specifically useful to repository administrators. Be careful with the **svnadmin**! Unlike **svnlook**, which is read-only, **svnadmin** has the ability to modify the repository.

The most-used feature is probably `svnadmin setlog`. A commit's log message is an unversioned property directly attached to the revision object; there's only one log message per revision. Sometimes a user screws up the message, and it needs to be replaced:

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

There's a nice CGI script in `tools/cgi/` that allows people (with commit-access passwords) to tweak existing log messages via web browser.

Another common use of **svnadmin** is to inspect and clean up old, dead transactions. Commits and updates both create transaction trees, but occasionally a bug or crash can leave them lying around. By inspecting the datestamp on a transaction, an administrator can make a judgment call and remove it:

```
$ svnadmin lstxns myrepos
319
321
$ svnadmin lstxns --long myrepos
Transaction 319
Created: 2002-07-14T12:57:22.748388Z
...
$ svnadmin rmtxns myrepos 319 321
```

Another useful subcommand: **svnadmin undeltify**. Remember that the latest version of each file is stored as fulltext in the repository, but that earlier revisions of files are stored as “deltas” against each next-most-recent revisions. When a user attempts to access an earlier revision, the repository must apply a sequence of backwards-deltas to the newest fulltexts in order to derive the older data.

If a particular revision tree is extremely popular, the administrator can speed up the access time to this tree by un-“deltifying” any path within the revision—that is, by converting every file to fulltext:

```
$ svnadmin undeltify myrepos 230 /project/tags/release-1.3
Undeltifying `/project/tags/release-1.3' in revision 230...done.
```

Repository Hooks

A *hook* is a program triggered by a repository read or write access. The hook is handed enough information to tell what the action is, what target(s) it's operating on, and who is doing it. Depending on the hook's output or return status, the hook program may continue the action, stop it, or suspend it in some way.

Subversion's hooks are programs that live in the repository's `hooks` directory:

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

This is how the `hooks` directory appears after a repository is first created. It doesn't contain any hook programs—just templates.

The actual hooks need to be named `start-commit`, `pre-commit` and `post-commit`. The template (`.tmpl`) files are example shell scripts to get you started; read them for details about how each hook works. To make your own hook, just copy `foo.tmpl` to `foo` and edit.

(The `read-sentinels` and `write-sentinels` are not yet implemented. They are intended to be more like daemons than hooks. A sentinel is started up at the beginning of a user operation. The Subversion server communicates with the sentinel using a protocol yet to be defined. Depending on the sentinel's responses, Subversion may stop or otherwise modify the operation.)

Here is a description of the hook programs:

`start-commit` This is run before the committer's 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 to 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.)² 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 a **`tools/hook-scripts/commit-access-control.pl`** script that can be called from **`pre-commit`** to implement fine-grained access control.

`post-commit` This is run after the transaction is committed, and we have a new revision. Most people use this hook to send out descriptive commit-emails or to make a hot-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 **`tools/hook-scripts/commit-email.pl`** script that can be used to send out the differences applied in the commit to any number of email addresses. Also included is **`tools/backup/hot-backup.py`**, which is a script that perform hot backups of your Subversion repository after every commit.

Note that the hooks must be executable by the user who will invoke them (commonly the user `httpd` runs as), and that same user needs to be able to access the repository.

The **`pre-commit`** and **`post-commit`** hooks need to know things about the change about to be committed (or that has just been committed). The solution is a standalone program, **`svnlook`** the section called “Examining a Repository”.) which was installed in the same place as the **`svn`** binary. Have the script use **`svnlook`** to examine a transaction or revision tree. It produces output that is both human and machine-readable, so hook scripts can easily parse it. Note that **`svnlook`** is read-only—it can only inspect, not change the repository.

²In this book, this is the only method by which hooks can implement fine-grained access control beyond what `httpd.conf` offers. In a future version of Subversion, we plan to implement ACLs directly in the filesystem.

Migrating a Repository (dump/load)

###TODO write this.

Adding projects

Examples of 'svn import'

###TODO write this.

Vendor Branches

(discussion of strategies and `svn_load_dirs.pl`)

###TODO write this.

Migrating a repository

Sometimes special situations arise where you need to move all of your filesystem data from one repository to another. Perhaps the internal fs database schema has changed in some way in a new release of Subversion, or perhaps you'd like to start using a different database “back end”.

Either way, your data needs to be migrated to a new repository. To do this, we have the **svnadmin dump** and **svnadmin load** commands.

svnadmin dump writes a stream of your repository's data to stdout:

```
$ svnadmin dump myrepos > dumpfile
* Dumped revision 0.
* Dumped revision 1.
* Dumped revision 2.
@hellip;
```

This stream describes every revision in your repository as a list of changes to nodes. It's mostly human-readable text; but when a file's contents change, the entire fulltext is dumped into the stream. If you have binary files or binary property-values in your repository, those parts of the stream may be unfriendly to human readers.

After dumping your data, you would then move the file to a different system (or somehow alter the environment to use a different version of **svnadmin** and/or `libsvn_fs.so`), and create a “new”-style repository that has a new schema or DB back-end:

```
$ svnadmin create newrepos
```

The **svnadmin load** command attempts to read a dumpstream from stdin, and effectively replays each commit:

```
$ 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) >>>
```

Voila, your revisions have been recommitted into the new repository.

And because **svnadmin** uses standard input and output streams for the repository dump and load process, people who are feeling saucy with Unix can try things like this:

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

Partial dump/load

You can also create a dumpfile that represents a specific range of revisions. **svnadmin dump** takes optional starting and ending revisions to accomplish just that task.

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

Now, regardless of the range of revisions used when dumping the repository, the default behavior is for the first revision dumped to always be compared against revision 0, which is just the empty root directory /. This means that the first revision in any dumpfile will always look like a gigantic list of “added” nodes. We do this so that a file like `revs-100-200.dumpfile` can be directly loaded into an empty repository.

However, if you add the `--incremental` option when you dump your repository, this tells **svnadmin** to compare the first dumped revision against the previous revision in the repository, the same way it treats every other revision that gets dumped. The benefit of this is that you can create several small dumpfiles 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 dumpfiles 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 dumpfile a new range of 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 dumpfile data for all the revisions that were added to the repository since the last time the script ran.

```
#!/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 = `svnadmin 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!
```

As you can see, the Subversion repository dumpfile format, and specifically **svnadmin**'s use of that format, 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.

Chapter 6. Advanced Topics

Run-time Configuration Area

When you first run the `svn` command-line client, it creates a per-user *configuration area*. On Unix-like systems, a `.subversion/` directory is created in the user's home directory. On Win32 systems, a `Subversion` folder is created wherever it's appropriate to do so (typically somewhere within `Documents and Settings\username`, although 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 preferred `$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 “global” 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 “All Users” area, a collection of global registry settings, or a collection of per-user registry 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-time settings 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 “metadata” 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-value by 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-separated array of values. (More about this later).

Property changes are still considered “local 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 “property” column 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-standard way 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 “magic” 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 <code>Date</code> . The keyword substitution of <code>\$LastChangedDate\$</code> will look something like <code>\$LastChangedDate: 2002-07-22 21:42:37 -0700 (Mon, 22 Jul 2002) \$</code> .
LastChangedRevision	The last revision in which this file changed. Can be abbreviated as <code>Rev</code> . The keyword substitution of <code>\$LastChangedRevision</code> will look something like <code>\$LastChangedRevision: 144 \$</code> .
LastChangedBy	The last user to change this file. Can be abbreviated as <code>Author</code> . The keyword substitution of <code>\$LastChangedBy\$</code> will look something like <code>\$LastChangedBy: joe \$</code> .
HeadURL	A full URL to the latest version of the file in the repository. Can be abbreviated as <code>URL</code> . The keyword substitution of <code>\$HeadURL\$</code> will look something like <code>\$HeadURL: http://svn.collab.net/repos/trunk/README \$</code> .
Id	A compressed summary of the other keywords, for example: <code>\$Id: bar 148 2002-07-28 21:30:43 epg \$</code> . This means the file <code>bar</code> was last changed in revision 148 by committer <code>epg</code> , 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-directories to 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 “magic” `svn:` property to a directory: the `svn:externals` property.

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

```
$ svn propset svn:externals projectdir
subdir1/foo      http://url.for.external.source/foo
subdir1/bar      http://blah.blah.blah/repositories/theirproj
subdir1/bar/baz  http://blog.blog.blog/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.

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`).

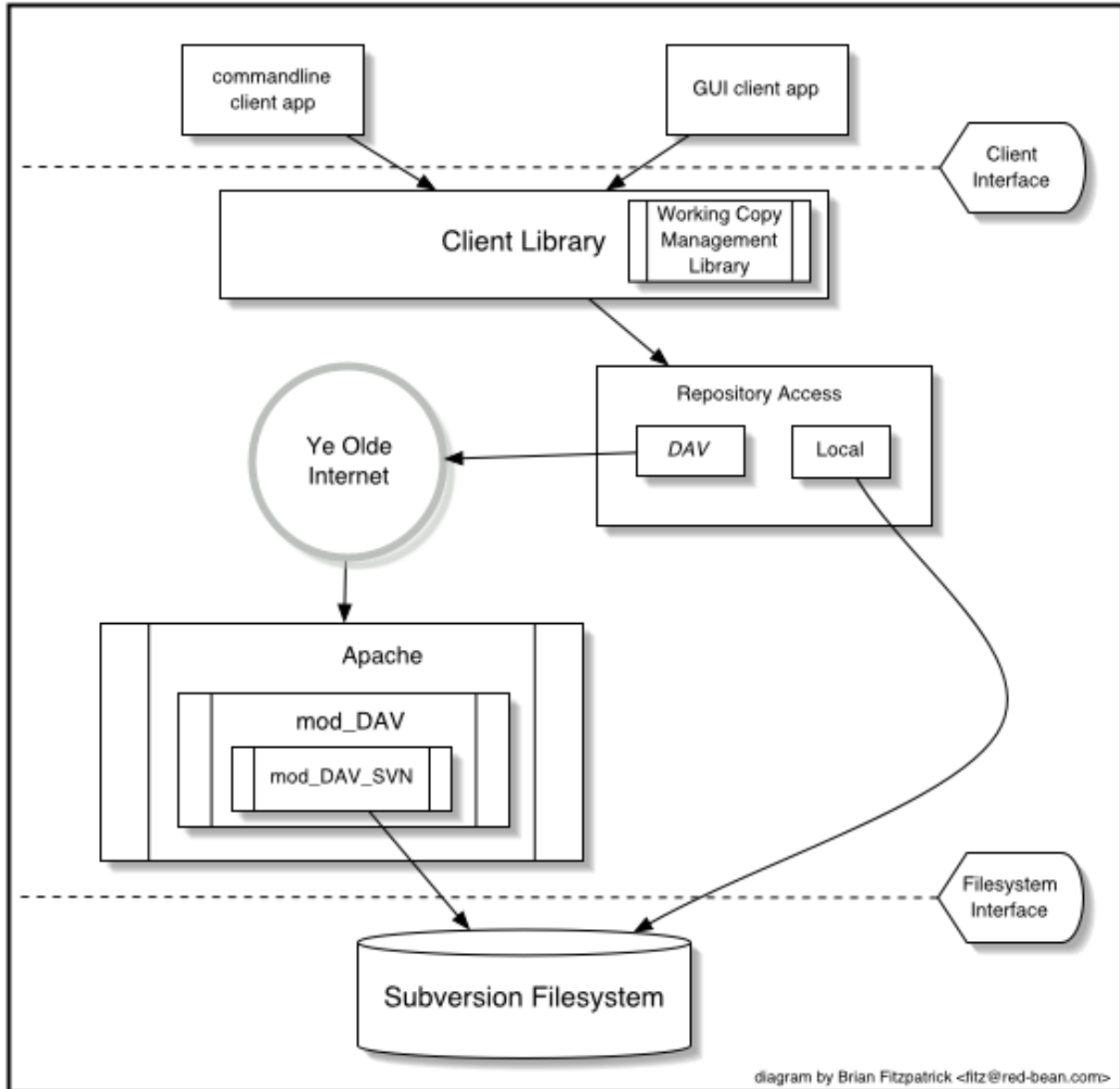
Table 7.1. A brief inventory of the Subversion libraries

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

The fact that the word "miscellaneous" only appears once in Table 7-1 is 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 “Client 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.³ 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 to 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).

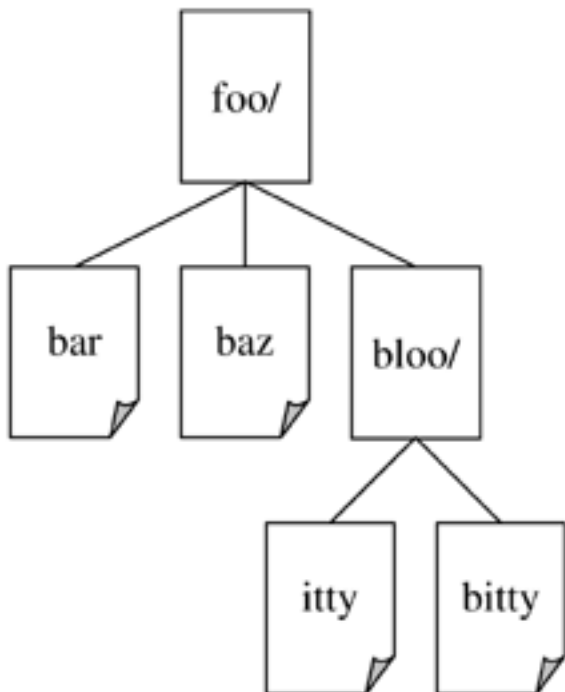
Most of the functionality provided by the filesystem interface comes as an action that occurs on a filesystem path. That is, from outside of the filesystem, the primary mechanism for describing and accessing the individual revisions of files and directories comes through the use of path strings like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

³The 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.

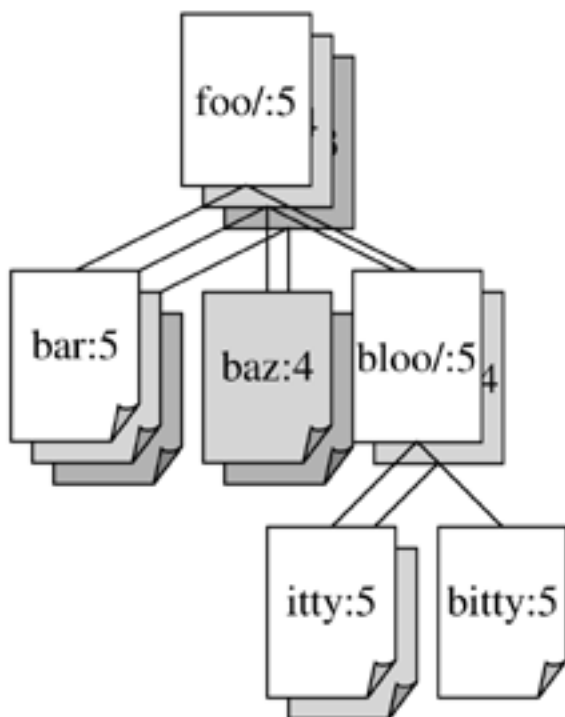
Figure 7.2. Files and directories in two dimensions



Of course, the Subversion filesystem has a nifty third dimension that most filesystems do not have—Time! ⁴ 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!

⁴We 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-successful error 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 "
               "%s" 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-world repository 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-source server 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-end to) `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-scale Subversion 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-encrypted communications 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-specified redirects (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 propagating 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 "Inside 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 de-

fined (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 interested in these naming conventions, but they definitely help to clarify the scope 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. Currently 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, 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 “Programming with Memory Pools” for more information.

URL and Path Requirements

With remote version control operation as the whole point of Subversion's existence, it makes sense that some attention 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 arguments 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, Subversion 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 Interface 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 speaking, 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

⁵Subversion 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-type language 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-based interface 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 reference counting and garbage collection, users of the language don't have to bother themselves with allocating and freeing 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 library 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-defined properties 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-known commit 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`-like file.

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 6 (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

⁶That 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-control concepts 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-purpose DeltaV 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-only resource. 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 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-level directory. 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'll need to grow comfortable with pools and how to wield 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-saving bit 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-mail based discussion lists are the primary communication media for Subversion development. See the Mailing Lists section of the website for other Subversion-related lists 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 include 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:`⁷

```
$ 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 privileges on the Subversion source repository are earned—a government by meritocracy.⁸ 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 thorough⁹ 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

⁷Note 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.

⁸While this may superficially appear as some sort of elitism, this "earn your commit privileges" 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.

⁹You 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 privileges (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-start guide for CVS users new to Subversion. It's not a substitute for real documentation and manuals; but it should give you a quick conceptual “diff” 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 “broken” 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 “revision 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 “revision 5 of `foo.c`”. Instead, one would say “`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 “pristine” 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 **cvstatus** 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 `f00/` 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 “revision 5 of directory `f00`” to mean a specific collection of directory-entries and properties. Now suppose we start adding and removing files from `f00`, and then commit. It would be a lie to say that we still have revision 5 of `f00`. However, if we bumped `f00`'s revision number after the commit, that would be a lie too; there may be other changes to `f00` 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 “perfect” revision of a directory.* Most of the time, your working copy will contain “imperfect” 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 “conflict 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 “Basic Workcycle” for more details. In particular, read the section about “Merging 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

“merged” during update, no real merge occurs. Instead, Subversion creates two files side-by-side in 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 “Modules”.

Branches and Tags

Subversion doesn't distinguish between filesystem space and “branch” 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.